Improving latency for interactive, thin-stream applications over reliable transport

by

Andreas Petlund

Doctoral Dissertation submitted to
the Faculty of Mathematics and Natural Sciences
at the University of Oslo
in partial fulfilment of the requirements for
the degree of Philosophiae Doctor

October 2009

Abstract

A large number of network services use IP and reliable transport protocols. For applications with constant pressure of data, loss is handled satisfactorily, even if the application is latency-sensitive [108]. For applications with data streams consisting of intermittently sent small packets, users experience extreme latencies more frequently [50]. Due to the fact that such *thin-stream* applications are commonly interactive and time-dependent, increased delay may severely reduce the experienced quality of the application. When TCP is used for *thin-stream* applications, events of highly increased latency are common, caused by the way retransmissions are handled [50]. Other transport protocols that are deployed in the Internet, like SCTP, model their congestion control and reliability on TCP, as do many frameworks that provide reliability on top of unreliable transport. We have tested several application- and transport layer solutions, and based on our findings, we propose sender-side enhancements that reduce the application-layer latency in a manner that is compatible with unmodified receivers. We have implemented the mechanisms as modifications to the Linux kernel, both for TCP and SCTP. The mechanisms are dynamically triggered so that they are only active when the kernel identifies the stream as thin. To evaluate the performance of our modifications, we have conducted a wide range of experiments using replayed *thin-stream* traces captured from real applications as well as artificially generated *thin-stream* data patterns. From the experiments, effects on latency, redundancy and fairness were evaluated. The analysis of the performed experiments shows great improvements in latency for *thin streams* when applying the modifications. Surveys where users evaluate their experience of several applications' quality using the modified transport mechanisms confirmed the improvements seen in the statistical analysis. The positive effects of our modifications were shown to be possible without notable effects on fairness for competing streams. We therefore conclude that it is advisable to handle thin streams separately, using our modifications, when transmitting over reliable protocols to reduce retransmission latency.

Acknowledgements

I would like to thank Dr. Carsten Griwodz and Dr. Pål Halvorsen for encouragement, ideas, feedback and enthusiasm in abundance. Their dedication and passion rubs off, making good days delightful and bad days endurable. I would also like to thank Espen Søgård Paaby, Jon Pedersen and Kristian Riktor Evensen for their important contributions to the work that has been done.

The working environment at Simula Research Laboratory is one of collaboration, support and friendly competition, which I find inspiring. I want to thank all colleagues and friends at Simula for input, feedback and discussions over the Friday cake. During the work on this thesis, I have worked with two of the most excellent companions one could wish for: Knut-Helge Vik has taught me the meaning of hard work as well as lightened the days with brilliantly dry humour. Håvard Espeland always has some interesting information to share, especially about Linux, FOSS and whisky.

Thanks to my fiancee, Anne, for love, patience, support. When I get too focused on computers, she puts my life back into perspective. Finally, I want to thank my parents for supporting me in all I decide to do.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The communicating citizen of today is present on Twitter, Facebook and the blogosphere. From the most remote location, she can stay connected to the pulsating world community. As a participant in conference calls and online games, she can work, play and interact. The infrastructure to support this connectivity is provided by the Internet. As the available bandwidth has multiplied in the last decades, the number of services provided over the Internet has exploded. Despite the rate of development, the Internet is still only providing a best-effort service, and loss and delay are frequently experienced. For interactive applications, such delays affect the quality of the service provided. We have studied a class of such distributed, interactive applications, focusing on identifying the reasons for the high experienced latencies that often occur. Using what we have learnt from this analysis, we have set out to reduce the latency for our target class of applications.

1.1 Background and motivation

Internet technology has undergone radical developments since its early ARPANET days. In the beginning, there were large challenges related to bandwidth and capacity. This led to research being focused on bandwidth sharing (fairness) and congestion avoidance. In the last couple of decades, we have seen tremendous developments in networking technology, resulting in much higher bandwidths. This development is accompanied by a tendency among Internet users in general to consume much more bandwidth, both for uploading and downloading. The increase in bandwidth consumption is accelerated by *peer-to-peer* technology like BitTorrent [33] and Spotify [94].

Parallel to the trend of increased bandwidth usage on the Internet, real-time communication applications have also evolved and gained ground. Presently, the Internet is used as medium for a wide range of interactive services like chat, remote desktop, stock trade systems, IP telephony and networked games. The element of interactivity, however, leads to latency requirements;

users become dissatisfied when they must wait for the system to respond. This is often problematic as the basic architecture of internetworking is based on best-effort services. Research has been performed, looking into ways of assuring a fixed quality of service (QoS) for data transport, but as the most successful approaches need support along the path of the connection [24, 21], such approaches have not yet gained ground. A consequence of the lack of QoS mechanisms is that we still have to rely on end-to-end approaches in order to provide data delivery across the Internet.

The most common end-to-end transport protocols today are the Transmission Control Protocol (TCP) [91, 17, 82] and the User Datagram Protocol (UDP) [90]. Other protocols that seek to extend the range of services and versatility are under development, like the Stream Control Transmission Protocol (SCTP) [95]. TCP is the prime choice for applications that need reliability and in-order delivery of data. It also provides congestion control, enabling the sharing of network capacity between concurrent streams. UDP is unreliable, but allows the sending application to determine the transmission timing. This makes UDP a common choice for time-dependent applications with no need for reliability. Even though UDP provides control of transmission timing, many interactive applications need reliability, at least occasionally, which then has to be implemented on the application layer (examples of frameworks that provide such services are the adaptive communication environment (ACE) [3], ENet [37] and UDP-based data transfer (UDT) [52, 14]). However, because of the lack of congestion control, and to avoid broadcast messages, UDP is frequently blocked by Internet Service Provider (ISP) firewalls. Consequently, many current time-dependent and interactive distributed applications are implemented using reliable transport protocols, such as TCP. In addition, many applications that use UDP despite its shortcomings, use a reliable transport protocol as fallback when UDP is blocked.

Many of the interactive applications commonly used today display patterns of transmission that deviate from the much researched greedy stream scenario where the application consumes as much bandwidth as possible. The data sent by such interactive applications are often small messages, and the timing is determined by the users' actions. A high-quality experience in, for example, a gaming scenario requires responses within 100-1000 ms depending on the game type [32]. The data patterns resulting from such event-based interaction are distinguished from greedy streams by high packet interarrival times (IATs) (i.e., a low packet rate) and small packet sizes. We call streams with such data patterns *thin streams*.

As the focus for reliable protocols has been on achieving higher throughput, mechanisms for recovering lost segments assume a steady supply of data from the application. Such steady supply is not provided by thin-stream applications, and because recovery relies on this, thin streams in an interactive scenario may therefore experience devastating delays. The inability of the currently used mechanisms of reliability and congestion control to provide low latency for thin-stream applications, and the development of solutions to lower the latency for the thin-stream scenario, are the focus of this thesis. Experimental protocols have been developed that may help to address thin-stream latency issues. We have, however, chosen to use the already deployed protocols (TCP and SCTP) as the basis for our investigations. By applying sender-side, standards compliant modifications to these widely used protocols, we aim to reduce thin-stream latency for endpoints that already support the said protocols.

1.2 Thesis context

This thesis is a part of the "Middleware Services for Management of Shared State in Large-Scale Distributed Interactive Applications" (MiSMoSS) project[1]. The aim of the MiSMoSS project was to improve support for distributed, interactive applications by abstracting systems requirements like group communication, latency hiding and network adaptation. The final goal was to create a set of services that could be presented to developers of distributed, interactive applications in order to improve the quality of their application. The MiSMoSS project had a three-part focus:

1. Latency-hiding for interactive applications: Strategies have been developed to make the observed effect of network latency as low as possible. This was done by employing techniques adapted to each application [79, 78, 80].

2. The effective construction of overlay networks for group communication: A wide range of algorithms was evaluated by simulation and experiments to find effective methods for group communication infrastructure [101, 102, 51, 103, 107, 105, 106, 104].

3. Latency reduction by well-adapted network techniques: To minimise the experienced delay by investigating network mechanisms that can provide an improved level of service [69, 84, 50, 54, 86, 87, 38, 85].

This thesis is part of item 3), and focuses on the latency-challenges that interactive applications experience when using reliable transport protocol. In our earlier work on this topic, we have experimented with QoS approaches, stream aggregation and transport protocol adaptation [69, 50]. Given the current state of Internet architecture, the QoS approach showed little promise. Stream aggregation, multiplexing many small game events destined to multiple users in the same stream to benefit from the greedy stream mechanisms in the intermediate hops of the data path, indicated great potential, but is reliant on an extended network of proxies. To

[1] This project was funded by the Norwegian Research Council under contract number 159992/V30. Parts of the work performed in this project has also been in the context of the information Access Disruptions (iAD) project under contract number 174867.

achieve a reduced latency for all participating hosts in a distributed, interactive scenario, the most promising approach was to focus on end-to-end transport.

1.3 Problem statement

Distributed, interactive, thin-stream applications that communicate over the Internet are common today. The data patterns they produce are a result of the actions or events that trigger each message. Such data streams have sporadic transmissions with relatively high packet IAT and often small packet sizes. This is in stark contrast to the transmission patterns of greedy streams, which try to move as much data as possible across the connection. Many time-dependent applications use reliable protocols either because they need the provided services or as a fallback when unreliable options are blocked for some reason. The effects of thin streams over reliable transfer, however, has not been thoroughly investigated. Based on the observations that we have made about thin streams, and with a focus on reliable transport, we have formulated the following hypotheses:

- **Hypothesis 1:** *Thin streams are very often a product of time-dependent and/or interactive applications.*

Examples of human interactions that produce thin streams are remote desktop connections, voice over IP (VoIP) and online games. Examples of other applications that can be time-dependent and produce thin streams can be stock-exchange systems and sensor networks. We do not assume that all thin streams are time-dependent, nor do we believe that all time-dependent applications produce thin streams. Our assumption is that a large share of thin streams represent interactive applications where timeliness is crucial to the users' experience.

- **Hypothesis 2:** *Retransmission mechanisms and congestion control mechanisms have been developed to maximise throughput, and may therefore cause higher retransmission latency when the transported stream is thin.*

The main rationale behind this statement is that retransmission mechanisms that can recover lost data segments quickly depend on frequent feedback from the receiver. This is a side effect of delaying retransmissions as long as possible in order to avoid retransmitting spuriously. In a scenario where transmissions are sporadic and far between, a minimal amount of feedback is generated to trigger such speedy recovery.

- **Hypothesis 3:** *It is possible to adapt existing retransmission and congestion control mechanisms to achieve lower latency for thin streams without jeopardising performance for greedy streams.*

If hypotheses 1 and 2 are true, there is a large potential for improving the latency for thin streams. We believe that this can be done by adapting the existing transport protocol mechanisms in such a way that there will be no performance-reduction for the well-proven solutions that are optimised for greedy streams.

- **Hypothesis 4:** *We can take advantage of the thin stream properties to achieve lower delivery latencies for the thin-stream applications or services.*

In addition to modifying the existing mechanisms, we think that alternative ways of improving thin-stream latency can be devised. This may be realised by taking advantage of characteristic traits observed in thin streams.

- **Hypothesis 5:** *Modifications to improve thin-stream latency can be implemented in such a way that unmodified receivers may benefit from them.*

The Internet consists of a huge amount of heterogeneous endpoints (devices with different operating systems). To update them all with modifications or new protocols is a difficult task. By making the modifications sender-side only, an unmodified endpoint can get benefits from a modified sender that provides a service known to produce thin streams.

As our focus was on the latency-challenges for thin streams in reliable protocols, there are avenues of investigation that have not been followed. The following approaches have been deemed outside the scope of this thesis:

- We have performed some investigations into unreliable transport protocols with application-layer reliability. There is, however, an inexhaustible supply of different application-layer approaches. We have therefore chosen to focus on the reliable transport protocols that are widely used for interactive applications.

- We have chosen to focus on modifications that can enhance the performance of existing, widespread systems. Experimental protocols that needs to be supported on all nodes in order to be deployed have therefore not been considered.

- The focus of our work has been on end-to-end approaches. Network-layer mechanisms that may reduce latency for interactive applications have therefore not been considered.

- QoS solutions that needs to be supported along the path of the connection have been considered [69]. We chose, however, to concentrate our efforts on the common transport protocols since this approach showed great potential for improvement.

1.4 Contributions

To prove the hypotheses formulated above, we have performed work consisting of analysis, implementation and evaluation. The main contributions are listed here:

Investigation of thin-stream properties: The investigation of *Hypothesis 1* demanded that we performed in-depth analysis of data patterns from a wide range of interactive applications. The analysis we performed confirms that a wide range of the time-dependent applications show thin-stream properties. The variation in packet IAT and sizes for the thin streams was analysed with respect to their effects on delivery latency. Typical packet sizes ranged from 100 to 400 bytes and IATs ranged from 30-600 ms.

Thin-stream latency analysis: To address *Hypothesis 2*, extensive analyses of latency for typical thin-stream scenarios were performed. From these analyses, we were able to determine that thin streams suffer from extreme latencies. In a trace from the massively multiplayer online game (MMOG) Anarchy Online [44], for example, we found acknowledgement latencies of up to 67 seconds. By studying the analysed traces in detail, we were able to determine the main reasons for the increased latencies. Furthermore, we performed experiments to determine whether some reliable transport protocol variations reduce delivery latency for thin streams. We identified TCP New Reno as the best alternative for reliable thin-stream transport, but we concluded also that none of the tested variations provides satisfactory latencies for thin streams.

Adaptation of retransmission mechanisms to reduce latency for thin streams: We implemented modifications to the existing retransmission mechanisms (in the Linux kernel) that reduce latency for thin streams. The mechanisms include timer reset corrections and a new RTO_{min} value for SCTP, fast retransmission after one dupACK as well as linear timeouts for both TCP and SCTP and a bundling mechanism for TCP. The mechanisms are dynamically switched on and off so that traditional mechanisms are used for greedy streams. The modifications were evaluated thoroughly through a series of experiments. In answer to the questions posed by *Hypothesis 3*, we found that the modifications are able to provide lowered delivery latency for thin streams.

Implementation of a bundling mechanism that takes advantage of small packet sizes in thin streams to reduce latency: We implemented a new mechanism that takes advantage of the fact that many thin streams have very small packet sizes. The bundling mechanism sends unacknowledged data segments with new packets, so as to avoid retransmissions. In many cases (like for Gigabit Ethernet), the minimum frame size is much larger than the packet sizes produced by thin-stream applications, making bundling possible with very little actual overhead. In answer to *Hypothesis 4*, the mechanism was evaluated through extensive experiments, and was found to significantly reduce delivery latency for thin streams.

Evaluation of transport protocols and our thin-stream modifications: We evaluated the described approaches for TCP, SCTP and UDP with application-layer reliability. All our modifications were designed to be transparent to the receiver in answer to *Hypothesis 5*; any unmodified (standards compliant) receiver can receive the benefit of a modified sender. Our findings show that we are able to reduce delivery latency for all the thin-stream scenarios we evaluated, especially the worst-case latencies that ruin the user experience are significantly reduced. In addition to the experiments performed to measure the latency of thin streams when using reliable transport, we performed surveys where users evaluated the effect of our mechanisms. All our results show that latency can be reduced significantly for thin-stream interactive applications by applying our mechanisms.

Evaluation of the impact of our modifications on per-stream fairness: As the implemented modifications apply more aggressive retransmission strategies when thin streams are detected, we evaluated also the effect of our modifications on competing streams (fairness). This evaluation showed that the modifications to the retransmission mechanisms do not affect fairness because the thin stream's congestion window stays below the minimum congestion window size. The bundling mechanism leads to increased packet sizes in certain scenarios, and therefore needs more resources. The number of sent packets, though, is not much higher since the bundling mechanism does not trigger extra transmissions.

The subject matter of this thesis has resulted in five publications in peer-reviewed journals and conferences [54, 86, 87, 38, 85]. Additionally, the interactive demonstration was exhibited at NOSSDAV 2008 [89] and the thin-stream mechanisms and the Linux implementation were presented at the Linux Kongress 2008 in Hamburg [88].

In general, we conclude that thin streams needs special handling when transmitted over reliable protocols to provide a satisfactory user experience. Our developed mechanisms greatly reduce latency for such thin streams in the target scenarios.

1.5 Outline

The thesis describes our thin-stream investigations from analysis of traces from interactive applications via implementation and experimentation to end-user evaluations. Here, we introduce each chapter and give a short description of the topics discussed.

- **Chapter 2** introduces the properties of thin-stream applications. A range of different time-dependent thin-stream applications is presented and analysed. We also present an analysis of latencies from a typical thin-stream application.

- **Chapter 3** describes reliable transport protocols, with a focus on mechanisms that affect latency for thin streams. We also evaluate different transport protocols to determine their performance for thin-stream latency.

- **Chapter 4** describes our modifications to reduce latency for thin streams. Both the basic principles behind the mechanisms and their implementation are described.

- **Chapter 5** presents the experiments we have performed to evaluate the effects of our thin-stream modifications. Laboratory tests were performed with different loss patterns, as well as Internet evaluations.

- **Chapter 6** presents the user-surveys that we conducted to evaluate the subjective effects of our thin-stream modifications. We also present an analysis of the hit-probability in a first-person shooter game with and without our mechanisms. Finally an interactive demonstration of the effects of our mechanisms is described.

- **Chapter 7** concludes this thesis by summarising our findings. We present a critical assessment of our work and discuss the most relevant tradeoffs and choices pertaining to this work. Finally, we outline topics for extending our work in the future.

Chapter 2

Thin-stream applications

Much networked data traffic today represents aspects of real life. We interact in virtual environments, chat, control remote computers and hold VoIP conferences. The data streams generated by such interactive applications are different from what we call greedy streams. While greedy streams try to move a given amount of data between two nodes as quickly as possible, many interactive streams generates sporadic packets that contain information pertaining to the user's actions. This kind of streams with small packets and relatively high interarrival time between each packet, we call *thin streams*.

Table 2.1 shows a selection of applications whose network traffic has been analysed. The identifying element for the thin-stream applications, in contrast to greedy streams, is that they all have small packet sizes and high interarrival time between the packets, and the stream often keeps those properties throughout its lifetime. In the following sections, we discuss the statistics gathered from the different scenarios presented in table 2.1.

2.1 Games

Massively multi-player online games (MMOGs) allow thousands of users to interact concurrently in a persistent virtual environment. For this to work, there are stringent latency requirements whose exact nature depends on the model of interaction, which again typically differs between game genres. In 2006, MMOGs constituted one of the largest entertainment industries, with a steady annual growth reaching 44 percent of gamers [100], exceeding 13 million online users [111]. Figure 2.1 shows the development of the estimated number of MMOG subscribers since the first games were launched. The growth of MMOGs has shown a steady rate and shows no signs of slowing down. Hence, we chose data traffic from networked games as a core example of thin-stream application classes. With respect to user satisfaction, games require tight timeliness, with latency thresholds at approximately 100 ms for first-person shooter (FPS) games, 500 ms for role-playing games (RPG) and 1000 ms for real-time strategy games

	payload size			packet interarrival time (ms)						avg bandwidth	
application	(bytes)							percentiles		used	
	avg	min	max	avg	med	min	max	1%	99%	(pps)	(bps)
Casa (sensor network)	175	93	572	7287	307	305	29898	305	29898	0.137	269
Windows remote desktop	111	8	1417	318	159	1	12254	2	3892	3.145	4497
VNC (from client)	8	1	106	34	8	< 1	5451	< 1	517	29.412	17K
VNC (from server)	827	2	1448	38	< 1	< 1	3557	< 1	571	26.316	187K
Skype (2 users) (UDP)	111	11	316	30	24	< 1	20015	18	44	33.333	37K
Skype (2 users) (TCP)	236	14	1267	34	40	< 1	1671	4	80	29.412	69K
SSH text session	48	16	752	323	159	< 1	76610	32	3616	3.096	2825
Anarchy Online	98	8	1333	632	449	7	17032	83	4195	1.582	2168
World of Warcraft	26	6	1228	314	133	< 1	14855	< 1	3785	3.185	2046
Age of Conan	80	5	1460	86	57	< 1	1375	24	386	11.628	12K
BZFlag	30	4	1448	24	< 1	< 1	540	< 1	151	41.667	31K
Halo 3 - high intensity (UDP)	247	32	1264	36	33	< 1	1403	32	182	27.778	60K
Halo 3 - mod. intensity (UDP)	270	32	280	67	66	32	716	64	69	14.925	36K
World in Conflict (from server)	365	4	1361	104	100	< 1	315	< 1	300	9.615	31K
World in Conflict (from client)	4	4	113	105	100	16	1022	44	299	9.524	4443
YouTube stream	1446	112	1448	9	< 1	< 1	1335	< 1	127	111.111	1278K
HTTP download	1447	64	1448	< 1	< 1	< 1	186	< 1	8	> 1000	14M
FTP download	1447	40	1448	< 1	< 1	< 1	339	< 1	< 1	> 1000	82M

Table 2.1: Examples of thin (**greedy**) stream packet statistics based on analysis of packet traces.

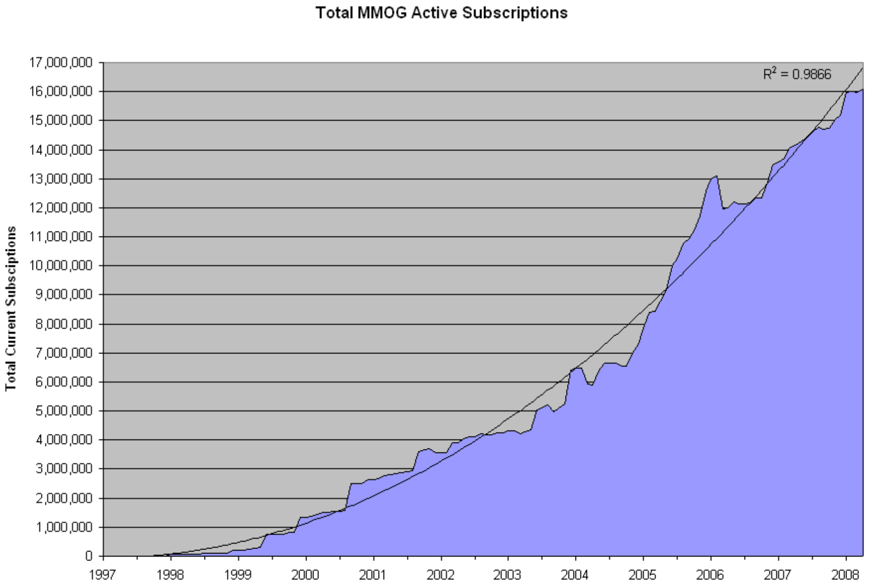


Figure 2.1: Estimated subscription development for MMOG games [111]

(RTS) [32]. With this in mind, supporting these kinds of games is challenging. The task is made even more difficult by the fact that a significant characteristic of this type of application is its lack of resilience towards network transmission delays [31].

We analysed packet traces from several games with regard to packet sizes and rates. Statistics from the traces are presented in table 2.1. The first is Funcom's popular role-playing

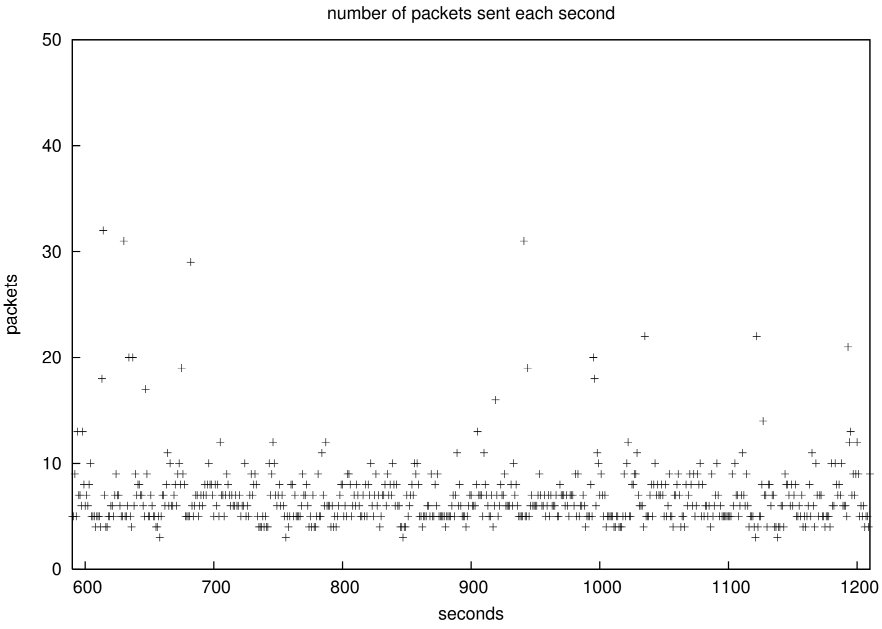


Figure 2.2: Packets per second for World of Warcraft

MMOG *Anarchy Online* (AO) [44]. The trace contains all packets from one of a few hundred game regions for about one hour. Less than one packet is sent per RTT on average, which means that the packet interarrival time is large (the average IAT is 632 ms). In addition, each packet contains only small game events, such as position updates. Thus, each packet is small (about 98 bytes payload on average). Since MMOGs are the paramount of networked gaming, we have analysed two more games, to illustrate the similarities of transmission patterns. World of Warcraft (WoW) [22] is the market-leading MMOG today with as much as 10 million active subscriptions in 2008 [111]. WoW traffic shows the smallest average payload size of all our examples. The packet interarrival time is lower than for Anarchy Online, but still high, leaving the throughput for the two games very close to each other. Figure 2.2 shows the number of packets per seconds for a selection of the trace (one stream). The plot shows how the stream keeps its low packet rate over time with the exception of a few events where the packet number increases somewhat. We have also analysed traces from a new MMOG, released in 2008: "Age of Conan" (AoC) from Funcom [43]. This trace shows slightly higher packet interarrival times, but still very small packets (80 bytes on average). The statistics from all three analysed MMOGs strengthens the impression that such interactive communication produces thin streams.

We have also included statistics for two games in the FPS genre: BZFlag [4] and Halo 3 [27]. Such games have higher intensity, and consequently more information that needs to be disseminated to the players. This results in relatively low interarrival-times between packets. The interarrival-times also vary with the intensity of gameplay within the given game (as the two Halo traces show). The packet sizes are small for both applications (very small for BZFlag).

FPS games have very strict requirements for timeliness, and players of FPS games will very quickly feel that latency degrades the quality of experience (QoE).

Finally, World in Conflict is analysed as an example of an RTS game. The traffic both to and from the server seems to be driven by an internal clock in the application, producing data with intervals of 100 ms. The payloads are very small in the client-server direction (4 bytes per packet on average), and somewhat larger in the server-client direction. This reflects the need to transmit the positions and actions of all the players to each client. World in Conflict can, based on this analysis, be categorised as a thin-stream interactive application.

2.2 Other thin-stream applications

Networked games are typical examples of thin-stream applications, but a wide range of other applications also displays the same network properties. The identifying factor is that transmission rate is limited by the application's production of data, not congestion control. We now present examples of thin-stream applications from a range of different areas where latency is an important factor.

2.2.1 Remote operation systems

When functions are to be remote-controlled, the operator issues commands, and responds to system feedback. This very often results in thin-stream data patterns. A characteristic of such control systems is that small packets with instructions are sent to the system being controlled, either periodically or in an event-based manner. Many of these systems operate in real time and require the system to react quickly to the control signals, i.e., a rapid packet delivery is a necessity.

Windows Remote Desktop using the remote desktop protocol (RDP) is an application used by thin client solutions or for remote control of computers. The analysed trace was from a session where mixed tasks like document editing and drawing of figures was performed. Analysis of packet traces indicates that this traffic clearly show thin-stream properties. The packet IAT averages 318 ms, and the packet size is 111 bytes on average. If second-long delays occur due to retransmissions, this results in visual delay for the user while performing actions on the remote computer.

Virtual network computing (VNC) is another common system for remote control of computers. In the analysed VNC session, a selection of tasks is performed including graphical editing and text editing. As more of the graphical content is transmitted using this system than for RDP, the difference in stream properties between client-server and server-client is large. The client-server traffic is very thin, while server-client has larger packets on average, but still relatively high interarrival times between packets.

Another way of working on a remote computer is by using the secure shell (SSH) protocol. This is used to create an encrypted connection to a remote computer and control it, either using text console, or by forwarding graphical content. The analysed dump presented in table 2.1 is from a session where a text document was edited on the remote computer. We can observe that this stream also displays the thin-stream properties. The interarrival times are very similar to the RDP session (323 ms on average), while the packet sizes are even smaller than for RDP.

2.2.2 Sensor networks

As an example of sensor networks we have analysed traffic from the real-time radar system in the Casa project [6], which performs research on weather forecasting and warning systems. Here, low-cost networks of Doppler radars are used that operate at short range with the goal of detecting a tornado within 60 seconds [114]. Control data between the server and a radar is typically small and sent in bursts. A packet trace (see statistics in table 2.1) shows that the average packet size from the server is 241 bytes, and a burst of four packets with an interarrival time of about 305 ms is sent every 30 seconds (the heartbeat interval of the system). To be able to detect a tornado in time, they rely on fast delivery of the radar measurement data.

Sensor networks monitoring other aspects of weather, traffic, waste dumps, greenhouses and so forth will in most cases display similar traffic patterns (due to semi-random triggering of events). Such monitoring systems are gradually becoming an integrated part of a modern society.

2.2.3 Audio conferences

Voice over IP (VoIP) with real-time delivery of voice audio data across the network is another example of a class of applications that produces thin data streams. There is a strict timeliness requirement due to the interactive nature of the applications. Nowadays, audio chat is typically included in virtual environments, and IP telephony is increasingly common. For example, many VoIP telephone systems use the G.7xx audio compression formats recommended by ITU-T. G.711 and G.729 which have a requirement of 64 and 8 Kbps, respectively. The packet size is determined by the packet transmission cycle (typically in the area of a few tens of ms, giving packet sizes of around 80 to 320 bytes for G.711) [55].

Skype [2] is a well-known conferencing service, with several million registered users, that communicate over the Internet. Table 2.1 shows statistics from the analysis of two Skype conferencing traces. We have compared the default behaviour of Skype using UDP to its fallback behaviour when using TCP. We can see that when Skype uses TCP, the packet sizes increase, as does the interarrival-time between packets. This makes for an increase in bandwidth, but not in the number of transmitted packets. The small average packet size combined with an interarrival time between packets that averages to 34 ms qualifies it as a thin-stream. To enable satisfactory interaction in audio conferencing applications, ITU-T defines guidelines for the one-way transmission time [59]. These guidelines indicate that users begin to get dissatisfied when the delay exceeds 150-200 ms and that the maximum delay should not exceed 400 ms.

2.3 Greedy streams vs. thin streams

Compared to the *greedy* streams shown in table 2.1, e.g., streaming a video from YouTube [99], downloading a document over hypertext transfer protocol (HTTP) from a server in the UK or downloading a CD-image from *uninett.no*, the examples given above are a small selection of applications whose data stream is *thin*. Other examples include virtual environments (such as virtual shopping malls and museums), augmented reality systems and stock exchange systems. All of these send small packets and have relatively low packet rates. Yet, they are still highly interactive and thus depend on the timely delivery of data.

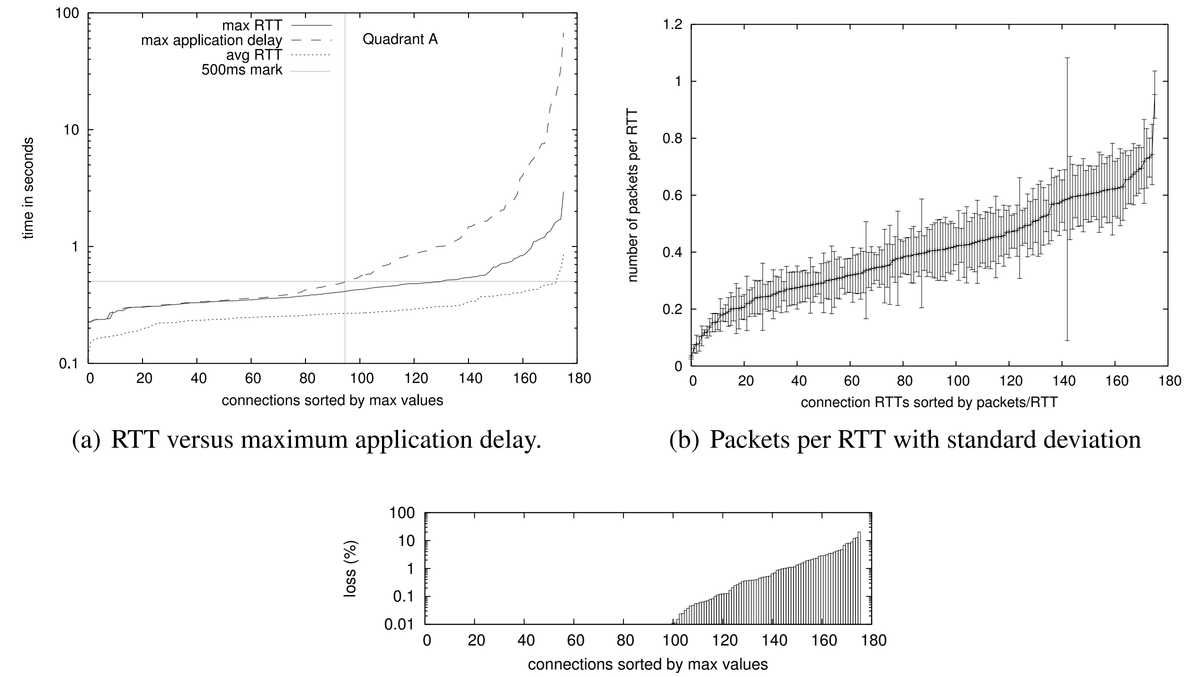
We have seen a range of examples of how interactive applications often produce data transmission patterns with small packet sizes and high interarrival time between packets. These patterns of transmission can, when combined with reliable transport protocols, lead to unwanted high latencies. We will now present a study where we analyse a typical thin-stream application with regard to latency.

2.4 Latency-analysis for an interactive thin-stream scenario

The design of reliable transport protocols has historically focused on maximising throughput without violating fairness[1]. As such, the retransmission mechanisms are attuned to high-rate applications like file transfers or bulk data streaming. We know of no previous studies that focus on the effect that thin-stream patterns have on delivery latency. Therefore, when we were given access to a server-side packet trace from *Funcom's* massively multi-player online game (MMOG) *Anarchy Online* [44], we performed a thorough analysis of the data to find out how frequent unwanted latency occurs. Server-client communication was the focus of our scrutiny, since we wanted to investigate server-side modifications that could benefit all clients on a modified server. Our analysis of thin-stream applications indicates, however, that the thin-stream properties are dominant both ways (server-client and client-server).

The *Anarchy Online* game world is divided into regions. Players that interact with each other in a specific in-game area are all connected to the same server. The connections are point

[1] We define fairness as per-stream TCP fairness. This means that greedy TCP streams that share a bottleneck achieves the same throughput over time, given that they have comparable network conditions (RTTs).



(a) RTT versus maximum application delay.

(b) Packets per RTT with standard deviation

(c) Per-stream loss rate.

Figure 2.3: Statistics from analysis of Anarchy Online server side dump [50].

to point TCP. The trace from Anarchy Online that we analysed, included about 175 separate TCP connections, representing players in one virtual game region hosted on a server physically located in the US.

Figure 2.3 shows loss- and delay statistics in a one-hour trace from the game server. In figure 2.3(a), we can see the average and maximum RTT and the maximum application-layer delay for each connection. We have marked a section of the plot (quadrant A) inside which all streams have experienced latency events that may severely degrade the QoE [32]. We can see from figure 2.3(b) that all of the connections have high interarrival time between packets, and qualify as thin streams. On average, all streams are well below 1 packet per RTT. The per-stream loss rate is shown in figure 2.3(c). Many of the analysed connections have no registered loss at all. Good connections or only short connection durations may explain this. For the connections that show loss, however, only very small amounts of loss are needed to cause high application delays[2]. The in-game experience of a delay of several seconds can be frustrating. When the delay exceeds 20 seconds ,as the statistics show several examples of, it may ruin the game session for the player. We have found no correlation between loss-events across the whole range of connections. We therefore assume that loss (for this dump) is not caused by server-side bottlenecks.

[2] The highest registered application delay for this dataset was ∼67 seconds. This was after 6 retransmissions were needed in order to recover the lost segment.

2.5 Summary

The analysis of interactive and latency-sensitive applications shows us that network patterns featuring small packets and high interarrival times are predominant. Table 2.1 presents a wide range of examples of interactive applications that display thin-stream properties. When transmitted using TCP, such thin streams are shown to exhibit aggravated latency when loss occurs. This is shown by the latency-analysis in figure 2.3. Retransmission mechanisms provided by reliable protocols that use TCP-based schemes seem to fail in providing acceptable delays when thin-stream traffic is retransmitted. In spite of the shortcomings of TCP in such scenarios, many developers still choose TCP for time-dependent applications. This is because of restrictive firewall policies that may stop alternative protocols, and the fact that reliability must be implemented on the application layer when using UDP. Newer, experimental protocols are not widely supported, and therefore very seldom used by commercial applications. To explain the reason why thin-stream applications are affected by latency events, we have to study the mechanisms that reliable protocols use to retransmit lost data and control congestion. To explain the high observed latencies when loss occurs, we describe the workings of reliable transport protocols next.

Chapter 3

Transport

As shown in chapter 2, thin streams can experience very high retransmission delays when loss occurs. The reason for the observed delays is to be found in the inner workings of the retransmission mechanisms of reliable protocols. When choosing a transport protocol for a time dependent application, the alternatives are limited to the following options:

1. Use established transport protocols (like TCP) that provide a range of services, but can yield high delays.

2. Use unreliable protocols (like UDP or DCCP) and implement reliability and in order delivery on the application layer.

3. Use an experimental reliable protocol that is tailored for the needs of time-dependent applications.

4. Use a quality of service (QoS) option.

However, QoS protocols have not become widely available, and the use of UDP has been criticised for its lack of congestion control. Consequently, many current time-dependent and interactive distributed applications are implemented using reliable transport protocols, such as TCP. In addition, many applications that use UDP despite its shortcomings, use a reliable transport protocol as fallback when UDP is blocked by a firewall. Also, when reliability is implemented on top of unreliable transport protocols (like UDP), the basic mechanisms of retransmission are often borrowed from TCP, yielding the same high thin-stream latencies. Experimental protocols are not widely supported, and therefore avoided by developers of commercial applications due to lack of availability at clients.

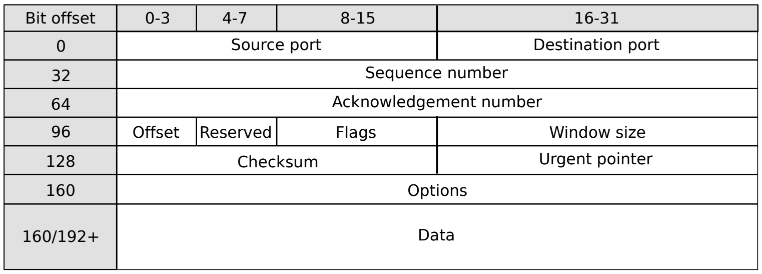
This chapter is devoted to describing the strengths and weaknesses of the different transport layer alternatives pertaining to thin streams. We describe TCP, SCTP, UDP and DCCP on the transport layer. We also present analysis of the performance of TCP (a range of versions), SCTP (Linux kernel SCTP) and UDP with application layer reliability when transmitting thin streams. Based on the results from thin-stream analysis we identify the principal reasons why thin streams experience extreme retransmission delays for reliable transfer.

3.1 TCP

One of the core components of the Internet protocol suite, TCP is one of the most used transport protocols on the Internet. Being the prime choice for reliable delivery used in for instance email, HTTP and FTP communication, it is widely supported and enabled for use in ISP firewalls. TCP is and end-to-end protocol; decisions pertaining to transmission are taken at the sender. Based on acknowledgements (ACKs) from the receiver, TCP tries to learn about the condition of the network path of the stream, and take appropriate measures. The basic services of TCP are as follows:

- Reliability: If data is lost, it is retransmitted until acknowledgement of successful delivery is received.
- In-order delivery: The stream of bytes is delivered to the receiver application in the same order as it was sent.
- Congestion control: If congestion is detected on the path of the stream, TCP adapts the send rate to allow concurrent TCP streams to share the bandwidth.
- Flow control: The sender does not transmit more data than the receiver has capacity to receive.
- Error control: By checksumming the transmitted data, transmission errors can be detected.

The services provided by TCP imply that the sender has to keep track of the current state of each TCP stream. It also means that a collection of data has to be transmitted with each sent packet in order to keep correct accounting of the TCP state at the sender and the receiver. This information is structured in the TCP header. The inclusion of a TCP header with every transmission results in a larger transmission overhead for TCP than for simpler transport protocols. Figure 3.1 shows the TCP header structure. Each TCP connection is uniquely defined by the IP address and the TCP port. The available capacity at the receiver is communicated through the “Window size”-field. The “Sequence number” is a counter that keeps track of the number of bytes the stream has transferred. The sender learns how much of the transmitted data has been successfully delivered by reading the “Acknowledgement number”. Other fields are used to keep track of the TCP state, setup and teardown process. Space is also reserved for custom options.



Bit offset	0-3	4-7	8-15	16-31
0	Source port			Destination port
32	Sequence number			
64	Acknowledgement number			
96	Offset	Reserved	Flags	Window size
128	Checksum			Urgent pointer
160	Options			
160/192+	Data			

Figure 3.1: TCP header structure.

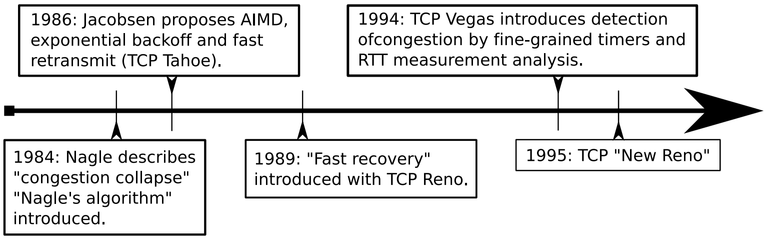

1986: Jacobsen proposes AIMD, exponential backoff and fast retransmit (TCP Tahoe).
1994: TCP Vegas introduces detection ofcongestion by fine-grained timers and RTT measurement analysis.
1984: Nagle describes "congestion collapse" "Nagle's algorithm" introduced.
1989: "Fast recovery" introduced with TCP Reno.
1995: TCP "New Reno"

Figure 3.2: Developments in TCP congestion control up to “New Reno” that are relevant to thin streams.

3.1.1 TCP developments culminating in TCP “New Reno”

After several stages of development, the first complete TCP draft [91] defined how to provide the basic services of TCP. One of the central tools for providing the services was a “go back N”-algorithm. “Go back N” enables the sending of several segments of data before receiving an acknowledgement. How much data to send before waiting for feedback was determined by the *receive window* specified by the receiver. The receive window size was usually defined by the buffer size for receiving data allocated by the receiving TCP code. Figure 3.3 illustrates an example of “go back N”-behaviour. The receiver advertises a window of 3 segments (it has capacity for 3 segments in its receive buffer). Segments 1 and 2 are successfully delivered, while segment 3 is lost. Retransmission of segment 3 is triggered by a timer, and normal behaviour is reestablished. Note that the receiver uses *cumulative acknowledgements* (cumACKs) signalling the successful reception of all data up to the acknowledged sequence number (index in the range of sent bytes). The use of cumACKs in early TCP also meant that all segments following a loss had to be retransmitted. Retransmissions were triggered by a timer which was based on a minimum value (1 second) and modified with an estimated RTT. Several RTT measurements were taken into consideration when calculating the retransmission timeout (RTO), resulting in a “smoothed” RTO.

When *John Nagle* described the “congestion collapse” in 1984 [76], he proposed two new features to be added to TCP. The first was dubbed “Nagle’s algorithm”, which aimed to avoid unnecessary sending of small packets by delaying transmission on the sender until a segment is full or a timer is triggered. Figure 3.4(a) shows one example of behaviour when Nagle’s

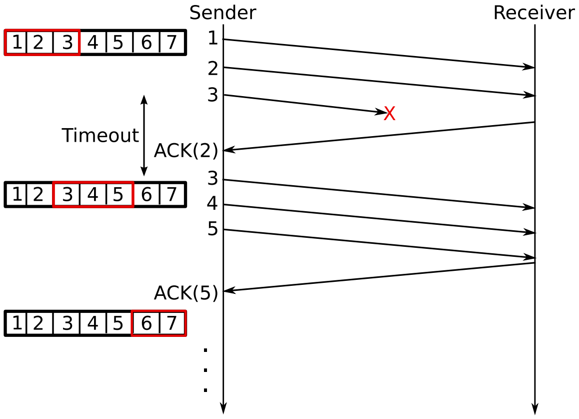
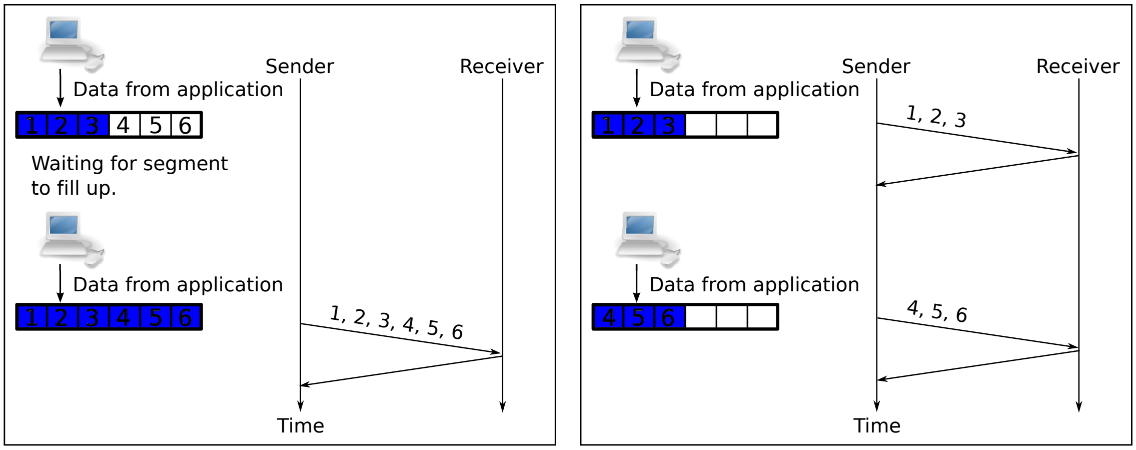


Figure 3.3: Example of "go back N" behaviour.



(a) With Nagle's algorithm. (b) Without Nagle's algorithm.

Figure 3.4: Example of packet transmission with and without Nagle's algorithm. A premise for the example is that there are unacknowledged data on the connection.

algorithm is active. The application delivers a small segment to be sent over TCP. The data is delayed in the network buffer until the data waiting to be sent can fill the maximum segment size (MSS). If no more data is received from the application within a given time limit, the smaller segment is sent. The data is only delayed if there are unacknowledged segments on the connection. The timing of transmissions for the same example without Nagle's algorithm is shown in figure 3.4(b). Here, the data is sent when delivered from the application, without further delay. Nagle also proposed a congestion control scheme using Internet control message protocol (ICMP) (source quench) packets to signal a reduction of the send rate before packets have to be dropped. This was later rejected since it generates unnecessary traffic.

In 1986, Van Jacobsen investigated occurrences of congestion collapse on the ARPANET. The investigations resulted in a paper where possible solutions to the recurring congestion problems were described [60]. The implementation of the new mechanisms in BSD 4.3 (Tahoe)

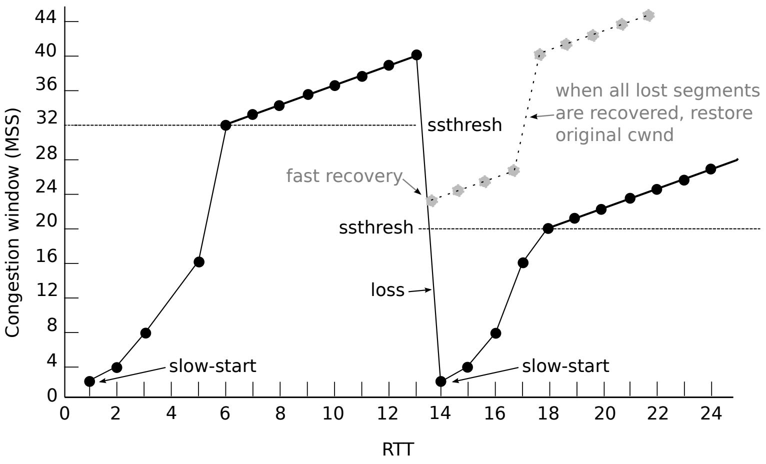


Figure 3.5: AIMD, slow-start and fast recovery example.

resulted in the first "modern" TCP, which added *congestion control* to the services provided by TCP. The introduction of the congestion window was made to limit the throughput in accordance with the available per-stream bandwidth.

Slow start was implemented to find the maximum available bandwidth within a short period of time. Congestion avoidance was instrumented as an "additive increase, multiplicative decrease" (AIMD)-algorithm. Figure 3.5 describes the AIMD-algorithm with slow start. The amount of sent data is doubled each RTT until the slow start threshold (*ssthresh*) is reached. At that point, the "additive increase" begins, incrementing the sent data by one MSS each RTT until loss is detected. A retransmission timeout is interpreted as a congestion notification, triggering the decrease. This causes the *ssthresh*[1] to be set to half the congestion window size, and slow start is initiated. In order to respond even more drastically to severe congestion, the *exponential backoff* mechanism was suggested. This mechanism doubles the retransmission timer for each successive retransmission of a given segment. This allowed for a total withdrawal of the competing streams upon severe congestion.

The "fast retransmit"-algorithm made for more effective retransmissions when feedback was abundant. If one of the segments in the send window was lost, but successive segments arrived at the receiver, the receiver would continue to send acknowledgements for the last in-order segment received (see figure 3.6). This effect of resending the last acknowledgement was called "duplicate acknowledgements" (dupACKs). After receiving three dupACKs, an assumption could be made that loss had occurred (not network reordering). The *ssthresh* would be set to half the congestion window size, a retransmission of the first unacknowledged segment would be made and TCP would enter slow-start. This allowed for quicker recovery than waiting for a retransmission timeout.

[1] The start value of *ssthresh* was, in early TCP versions, set to the size of the receive window. This was later changed so that ssthresh would reflect the estimated link capacity.

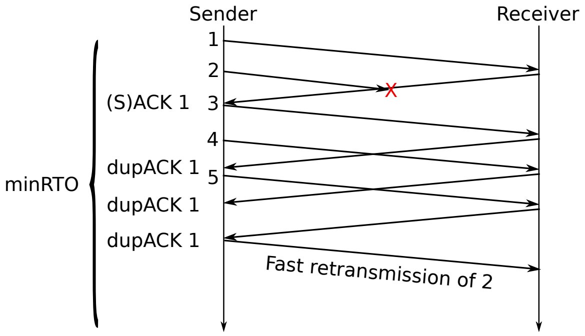


Figure 3.6: Example of how a fast retransmission is triggered after receiving three dupACKs.

With the introduction of “fast recovery” in 1989 [23], TCP Reno was born. When a fast retransmit was triggered, TCP would not go into slow start, but halve the congestion window, and continue to send data segments (shown with grey dots and dotted lines in figure 3.5). If all lost segments were recovered (acknowledged) before a timeout was triggered, the old congestion window would be restored, and transmission could continue at the previous rate. This improvement helped improve bandwidth utilisation since the streams would not all go into slow start when sporadic loss occurred.

Further improvement to TCP Reno was made in 1995 when a scheme to use ACK information to detect multiple holes in the ACK sequence was devised. When loss was detected by three dupACKs, the *ssthresh* would be set to half the congestion window (*cwnd*), and fast recovery would be initiated. The improved algorithm would, however, transmit a new segment from the end of the congestion window each time a dupACK was received, thus potentially maintaining a stream of ACKs. Upon reception of a dupACK that acknowledged some, but not all of the sent packets in the congestion window, it was assumed that a new hole was detected, and a new fast retransmission was performed. The fast recovery phase was kept until all holes had been “plugged” (or a timeout occurred) [41]. This TCP-version was branded “TCP New Reno”, and was, until very recently, the default TCP variation in many operating systems.

3.1.2 Retransmission timeout calculation

A critical part of packet recovery in TCP is how the retransmission timeout is calculated. The effect of this calculation is also to a large degree influencing the retransmission latencies we observe in our analysed traces.

The recommendations for calculating the RTO timer in TCP are summarised in RFC 2988 [82]. The RTO calculation is based on two intermediately calculated variables: Smoothed RTT (SRTT) and RTT variance (RTTVAR). SRTT and RTTVAR are calculated based on successful RTT measurements that are made as ACKs arrive. The specification says that Karn’s algorithm [61] must be used for RTT measurements. Karn’s algorithm states that RTT measurements from retransmitted data segments should never be used as the basis for RTO calculation. This is because the result is ambiguous when considering which transmission triggered the ACK used as basis for the calculation.

$$\begin{aligned} K &= 4 \\ SRTT &= R \\ RTTVAR &= \frac{R}{2} \\ RTO &= SRTT + max(G, K \times RTTVAR) \end{aligned} \tag{3.1}$$

Upon initiating a new connection, the RTO should be set to 3 seconds [23]. When the first RTT measurement R is made, the SRTT, RTTVAR and RTO should be set according to the algorithm shown in equation 3.1. G represents the timer granularity for the TCP implementation. K is set to the value specified in [82]. The specification states that a lower granularity ($G \leq 100ms$) yields more accurate RTT measurements. Another consideration pertaining to timer granularity is that the maximum wakeup time for a timeout is $RTO + G$.

$$\begin{aligned} \alpha &= \frac{1}{8}, \beta = \frac{1}{4} \\ RTTVAR &= (1-\beta) \times RTTVAR + \beta \times |SRTT - R'| \\ SRTT &= (1-\alpha) \times SRTT + \alpha \times R' \\ RTO &= SRTT + max(G, K \times RTTVAR) \end{aligned} \tag{3.2}$$

When subsequent ACKs arrive, the RTTVAR, SRTT and RTO have to be recalculated [82]. The variables are updated according to the algorithm in equation 3.2. α and β are set according to [60]. The new RTTVAR is calculated based on the current RTT measurement R', and the previous SRTT and RTTVAR. SRTT is computed based on the current RTT measurement and the previous SRTT value. The RTO value is set based on the newly calculated SRTT and RTTVAR. If the measured RTT fluctuates, the $K \times RTTVAR$ inflates, and the RTO becomes high. The rationale for this algorithm is that the RTT variations are indications of congestion. It therefore ensures a conservative RTO by making RTTVAR count to a large degree. This approach does, however, increase the retransmission latency, which has led to alternative RTO calculations being used some operating systems (like Linux).

If the calculated RTO is lower than 1 second, the value should be rounded up to one second according to the RFC. In several operating systems (for instance Linux), this minimum RTO (RTO$_{min}$) is kept at a lower value[2] to avoid unnecessary high retransmission delays. When the

[2] The Linux kernel (2.6.30) has an RTO$_{min}$ of 200 ms while FreeBSD 7.2 allows a 30 ms RTO$_{min}$.

```
        icsk->icsk_rto = min(((tp->srtt >> 3) + tp->rttvar), TCP_RTO_MAX)
```

Figure 3.7: The Linux kernel (2.6.23.8) code for calculating the RTO timer.

RTO$_{min}$ is chosen, it is a tradeoff between the need to avoid spurious retransmissions[3], and the need to keep retransmission delays low. The specification also states that a maximum value (of at least 60 seconds) may be placed on the RTO.

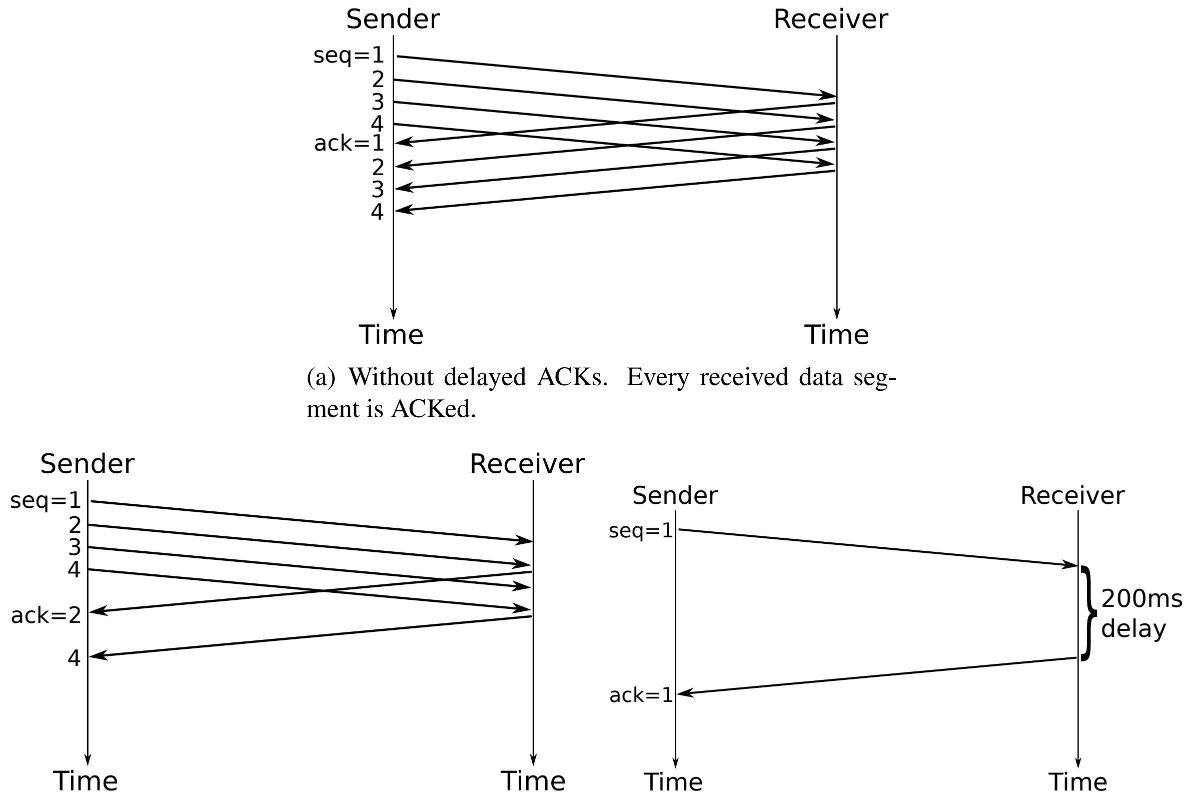
Figure 3.7 shows, as an example, the Linux kernel code for calculating the RTO. The variable *tp->srtt* holds the $SRTT \ll 3$, the current RTT variation is kept in *tp->rttvar* and *TCP_RTO_MAX* is the system's maximum allowed RTO. The effect is that the RTO is set to the $SRTT + RTTVAR$. $RTTVAR$ has a lower bound of 200 ms in the Linux kernel, effectively setting the RTO$_{min}$ to 200 ms. $SRTT$ and $RTTVAR$ is calculated as described in figures 3.1 and 3.2. The difference in the RTO calculation is made to tone down the effect of $RTTVAR$, thus making the RTO less vulnerable to fluctuations in the estimated RTT.

3.1.3 Delayed acknowledgements

Cumulative acknowledgements are the basic way of confirming packet delivery in TCP. The receiver inserts the sequence number of the highest in-order data segment that has been delivered into the ACK. The sender then knows that all sent segments below the given sequence number have been delivered. This technique opens for rationalising the delivery of ACKs. A common way of reducing the number of ACKs that are transmitted upstream is to use a mechanism called *delayed acknowledgements* (delayed ACKs). Figure 3.8 shows an example of how this technique works. Instead of sending an ACK for every received packet as shown in figure 3.8(a), the receiver waits for the next packet to arrive before sending a cumulative ACK that covers both segments (illustrated in figure 3.8(b)). If no further segments arrive, the ACK is triggered by a timer as shown in figure 3.8(c). RFC 1122 [23], which describes requirements for Internet hosts, states that TCP should implement delayed ACKs, but that the delay should not exceed 500 ms. A common value for the delayed ACK timer is 200 ms (used for instance in the Linux kernel). In addition to reducing the upstream traffic, the delayed ACKs can help reduce processing overhead for TCP since less packets have to be generated.

Delayed ACKs may present problems for TCP variations that rely on precise RTT measurements for its mechanisms because less feedback is available to get accurate RTT measurements. The combination of Nagle's algorithm and delayed ACKs can also cause high delays that may

[3] Spurious retransmissions are retransmissions that prove redundant in the process of recovering a lost segment. If, for instance, the RTO$_{min}$ is very low, a timeout may trigger a new retransmission even though the transmitted segment was successfully received and an ACK is underway to the sender. Generally, if throughput is the only relevant metric, spurious retransmissions are a waste of link resources.



(a) Without delayed ACKs. Every received data segment is ACKed.

(b) With delayed ACKs. Bandwidth is saved on the upstream path.

(c) With delayed ACKs. If no further segments arrive, the ACK is triggered by a timer.

Figure 3.8: Examples with and without delayed ACKs.

be unfortunate for time-dependent applications. This is because transmissions are delayed both on sender and receiver.

3.1.4 TCP Vegas

TCP Vegas [25] was a modification of "Reno" that was introduced in 1994. Using fine-grained timers, this TCP variation is able to detect congestion based on RTT measurement analysis and dynamically calculate the RTO. This enables retransmissions of segments before a "traditional" timeout would occur.

TCP Vegas differs from Reno in both retransmission scheme and congestion avoidance. The changes in the retransmission mechanisms are as follows:

- TCP Reno uses coarse-grained timers[4] to calculate RTT and variance. This reduces the accuracy of the calculated values and also influences the triggering of timeouts. The finer timer used in Vegas reduces the overhead when calculating timeouts.

[4] In TCP Reno (the BSD implementation), the granularity of the timer used to compute RTT and variance estimates is 500 ms [25]. This helps to keep TCP conservative when retransmitting, but can result in inaccurate RTT estimates, and makes the RTO prone to extra delays due to late checking.

- The fine-grained timers used to calculate RTTs based on system clock timestamps are used to retransmit in the following cases:

    1. When receiving the first dupACK, if the new RTT is greater than the RTO, the segment is retransmitted without waiting for 3 dupACKs to arrive.
    2. The first and second ACK after a retransmission are checked to see if the RTT is greater than the RTO. If the test is positive, a retransmission is performed.

  Vegas also uses coarse-grained timers (TCP Reno-style) in case the listed special cases do not identify a lost segment.

- TCP Vegas does not allow the congestion window to be halved more then one time in the course of one RTT. This is necessary since Vegas reacts more quickly to signs of congestion.

TCP Vegas also takes a different approach to the detection of loss and the congestion avoidance algorithms. While Reno detects loss and reacts to it, Vegas tries to detect signs that congestion is about to occur, and react before loss can happen. The following algorithm is used when slow-start is not in effect:

- Define a “baseRTT” for each segment to be sent. The value chosen for “baseRTT” is commonly the minimum observed RTT. Calculate an “expected throughput” (current congestion window size / “baseRTT”).

- Find the estimated “actual sending rate” by recording the number of bytes transmitted between the segment is sent and its ACK is received, and divide this number by the sample RTT. Do this calculation once per RTT.

- Compare the “actual throughput” to the “expected throughput” and adjust the window size based on this difference.

When in slow-start, TCP Vegas doubles its congestion window every second RTT (in contrast to each RTT for Reno). The less aggressive mechanisms of Vegas ensure better bandwidth utilisation in an all-Vegas environment. When competing with Reno (and other, more aggressive TCP flavours), TCP Vegas reacts earlier to congestion than its competing streams, and would receive less than it’s share of the available bandwidth [25][5].

[5] Windows Vista’s “Compound TCP” is reported to share similar properties to Vegas (window expansion by RTT measurements). It is also shown to produce slightly lower goodput than comparable TCP variations under many circumstances [56].

3.1.5 Selective Acknowledgements

One inherent limitation of using cumulative ACKs is that only assumptions can be made to which segments have been delivered and which have been lost. This problem was addressed by the “selective acknowledgement” (SACK) TCP extension [70]. When SACKs are in effect, the receiver uses TCP header extensions to report which segments are lost, and which are successfully delivered. The sender can, using the SACK information, retransmit only the lost segments, thus saving resources. SACKs have to be supported both by the sender and receiver, and the use of SACKs is negotiated in the three-way handshake.

3.1.6 Duplicate SACK

Duplicate SACK (DSACK) [42] is an extension to SACK that enables the identification of segments that have been transmitted more than one time. The receiver includes within the ACK the sequence number(s) of the packet(s) that triggered the ACK. The DSACK information can be used by the sender to better assess the network loss by identifying spurious retransmissions. The algorithm assumes that duplicates are caused by spurious retransmissions. If network duplication is detected (a DSACK can not be coupled with a retransmission), the algorithm is disabled.

3.1.7 Forward acknowledgements

TCP forward acknowledgement (FACK) [71] implements an algorithm where the SACK information is used to better estimate the amount of outstanding (sent, not ACKed) data. This is done by interpreting the highest SACKed sequence number as a sign that all lower unSACKed segments are lost. This helps to reduce burstiness in transmissions since it controls the outstanding data more accurately. The reordering of packets on the network does, however, break the algorithm, and FACK is therefore often disabled for a stream if reordering is detected.

3.1.8 Congestion control for high-speed links

A lot of research has been done to develop TCP mechanisms that can cope with high-speed links with loss. When the available bandwidth is generally high, AIMD has problems reaching Gbps speeds if there are occasional losses. On wireless links that experience relatively high link-layer delays (due to link-layer loss and retransmissions), reaching a proper end-to-end transmission rate is also a challenge. The research in this field has focused mainly on a flexible and speedy adaptation of the congestion window. The RTO and fast retransmit principles are usually affected only to a small degree (for instance by alternative methods for RTT estimation). The following TCP variations are central in the line of *cwnd*-adapting mechanisms.

TCP binary increase congestion (BIC) [113] is developed for networks with high bandwidth-delay products (BDPs). The congestion control algorithm can be categorised into three parts: 1) Additive increase, 2) binary search and 3) max probing. The key feature of BIC is that it searches for the middle point of two values: win_{max} and win_{min}. win_{max} is the window size when the last loss was detected (or a fixed value at the beginning of a connection). win_{min} is the window size after a reduction. If an increase in window size is made without any loss-events, win_{min} is set to the current window size, and a new search is performed. If the search returns a value that is larger than a given constant s_{max}, BIC increments the window by s_{max}. This gives the algorithm a linear growth in the beginning. The binary search yields smaller and smaller increments until it stabilises. If loss occurs, the window is reduced, and the search process starts again. This gives the algorithm the pattern of a binary search and it is able to quickly reach a suitable size for the congestion window. The search stops when the increase is less than a constant (s_{min}). When the window size reaches win_{max}, BIC enters “max probing” where it tries to increase the window size as long as no loss events are detected. The probing starts very slow, escalating to linear increase if successful.

In a revised version of BIC (TCP CUBIC) [53], the three-part window growth function is replaced with a cubic function. The cubic function is less aggressive near the equilibrium, and thus more fair to other streams (especially for connections with low RTT). The modification also simplifies window size calculation since three different stages are replaced with one function. A “TCP mode” is also incorporated to compensate for the parts of the cubic function which grows more slowly than a generic AIMD TCP variant (like New Reno) would. CUBIC is the choice TCP variation for several Linux distributions today.

TCP Westwood [30] was designed to improve throughput for connections with high capacity, tendency for loss and dynamic bandwidth (as often found in wireless networks). The central idea in Westwood is to use a bandwidth estimate to adjust *ssthresh* and the *cwnd*. An eligible rate estimation is calculated by counting the amount of acknowledged data over a time interval. The time interval is computed using TCP Vegas-like differences between estimated and actual rate (based on RTT measurements). This scheme aims to separate loss caused by noise or network layer delays from loss by congestion and treat each case separately.

Tests performed on the Westwood TCP variation soon revealed that the algorithm overestimated the available bandwidth (due to ACK compression[6]). This overestimation can cause Westwood to get a fairness-advantage over competing streams using other TCP variations. Westwood plus [48] was developed to counter this overestimation. The Westwood algorithm is modified so that a sample bandwidth is calculated every RTT instead of on ACK arrival. The bandwidth samples are spread evenly over the estimated RTT period, thus filtering events of

[6] ACKs have to arrive at the sender with the same inter-ACK spacing as they were sent in order to facilitate accurate *cwnd* calculation. When queueing happens on the feedback path, ACKs may arrive with small spacing, leading to an overblown *cwnd*

high ACK density.

Using regular AIMD, high bandwidths are near impossible to achieve[7]. This led to the development of "high-speed TCP" (HSTCP) [40]. In HSTCP, the *cwnd* calculation is done differently depending on the current size of the *cwnd*. When the window is small, HSTCP behaves like regular AIMD TCP variations. A large *cwnd* causes HSTCP to increase faster and decrease slower than regular AIMD (the parameters set based on the current *cwnd* size). HSTCP may react more slowly to congestion events and network changes (like new flows) [66], and thus grab more bandwidth than the competing streams when this happens.

Another TCP variation for high BDP connections is "scalable TCP" (STCP) [62]. The method suggested in STCP for adapting to high-speed networks is to use a constant to increment the *cwnd* ($cwnd = cwnd + 0.01$ for each received ACK). When the *cwnd* is large, this gives a much quicker growth than regular AIMD-based algorithms ($cwnd = cwnd + (\frac{1}{cwnd})$ per received ACK). To avoid unfairness towards competing non-STCP streams, a "legacy window" limit is defined. Thus, STCP uses regular AIMD window growth until the legacy window is reached and the modified growth function is activated. The use of a constant for *cwnd* calculation makes STCP simpler to implement than, for instance, the parametrised HSTCP. When the *cwnd* is large, competing streams may lose throughput due to the higher aggressiveness of STCP.

H-TCP [66] resembles HSTCP and STCP in its *cwnd* calculations, but aims to be more fair towards other competing streams. This is achieved by defining a time interval after each loss event, in which H-TCP adheres to the AIMD *cwnd* increase function. When the time interval has elapsed, the faster increase function is reinstated. This allows for fairness in "low-speed regimes", while enabling high throughput in "high-speed regimes".

TCP variations that adjust the *cwnd* each time an ACK arrives penalises streams with high RTTs. TCP Hybla [28] aims to remove this negative effect for streams with high RTTs. This is achieved by normalising the different streams' RTTs relative to a reference RTT (RTT_0). The normalised RTT is used to make the actual throughput independent of the RTT. Hybla also includes the SACK option since multiple losses are more probable over long RTTs, and multiple segment recovery therefore is sensible. High RTT connections may suffer badly from exponential backoff since an ACK from a non-retransmitted packet is needed to recover. In Hybla, exponential backoff is therefore not used.

A range of other high-speed TCP variations have been devised, but they all focus on flexible and rapid adjustment of the *cwnd*. Since thin streams does not depend on the *cwnd* fluctuations, but on timers and fast retransmit adjustments, they are not markedly affected by such variations.

[7] On a connection with packet sizes of 1500B and an RTT of 100ms, a loss rate of $\frac{1}{5\times10^9}$ would make it impossible to achieve an average rate of 10Gbps [40]. This is because the growth rate is too slow to catch up with the reduced congestion window (caused by loss) for such large window sizes.

3.1.9 Summary

We have described the basic mechanisms of TCP with focus on the mechanisms that affect retransmission delays for thin streams. Different approaches for timer calculations influence the retransmission delays when timeout occurs. TCP Vegas actively applies trusted RTT measurements, which is used to adjust retransmission timers. This could affect the observed retransmission delays for thin streams. Different strategies for triggering fast retransmissions affect the latency. Explicit gap notification like SACKs also potentially affect the retransmission latency if the sender uses the information to modify its retransmission schemes.

3.2 Evaluation of TCP retransmission delays

To evaluate the performance of available TCP variations in the context of thin streams, we have performed a range of tests where the focus of our analysis is retransmission latency. Results from analysing the *Anarchy Online* [50] concluded that thin streams suffer unwarranted high latencies when using TCP. This realisation led to another question: Is any of the currently available[8] TCP variations able to support thin streams? Experiments performed in [50] and extended in [77] show how different TCP congestion control mechanisms perform for thin-stream traffic.

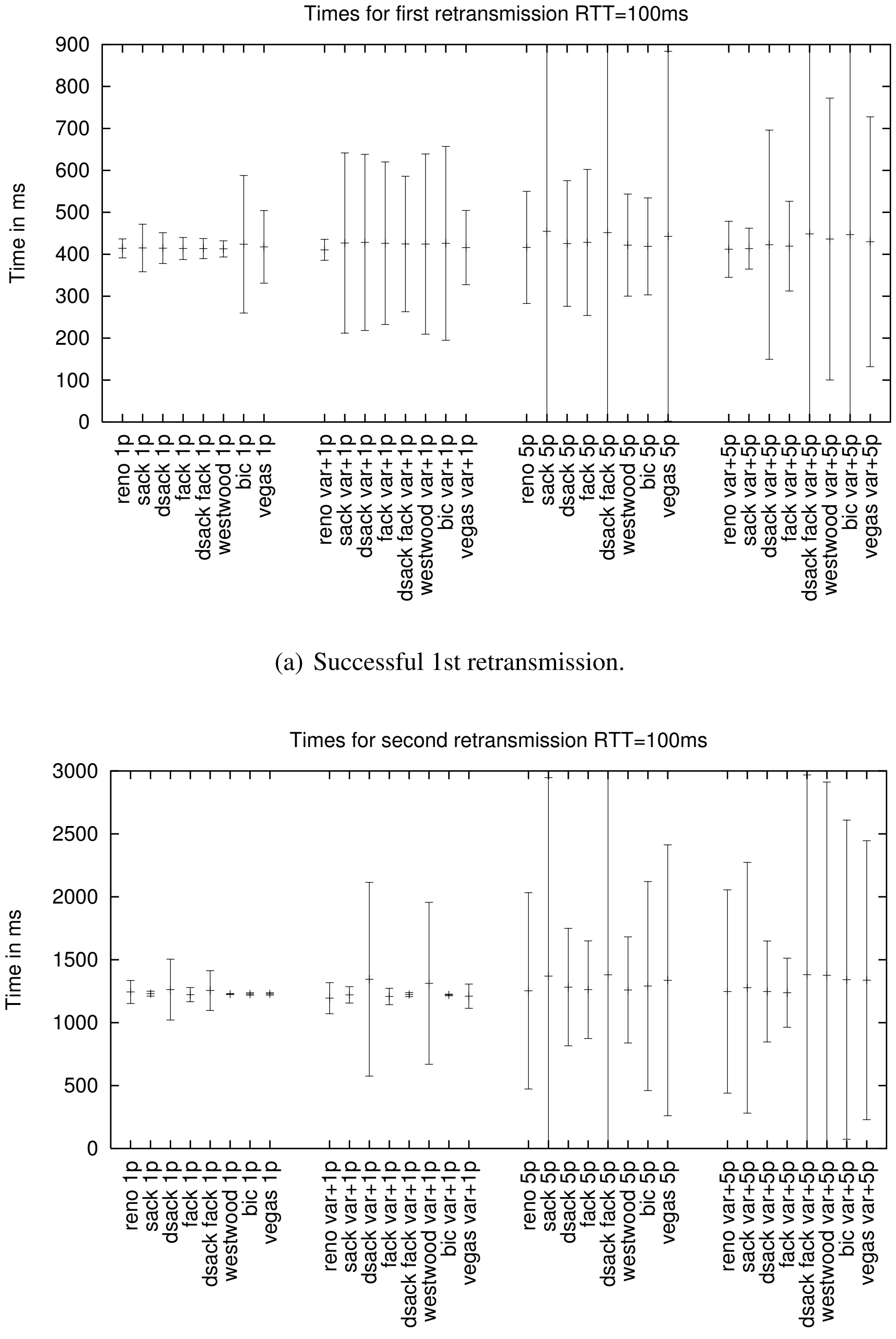
In [77], the following TCP variations were tested : Reno, BIC, high-speed TCP, H-TCP, Hybla, S-TCP, Vegas (Linux version without fine-grained timers) and Westwood. All TCP variations were tested with different combinations of SACK, DSACK and FACK in order to determine the influence of each mechanism on thin-stream latency. One of the findings was that only small differences in latency could be found between the TCP variations[9]. All the tested variations showed the same symptoms of high latency upon retransmissions.

Even though the differences between TCP variations were small, a tendency could be found that TCP New Reno performs best for a range of different parameters in the thin-stream scenario. Figure 3.9 shows the latency for a test setup over an emulated network (see figure 3.10) with an RTT of 100 ms. Traffic patterns for the test were generated by replaying the *Anarchy Online* trace that was analysed in section 2.4. The bars labelled "1p" have been exposed to 1% loss, while "5p" signifies a loss rate of 5%. In addition, 10% delay variation (jitter) has been added for the bars labelled "var+".

The results for the first retransmission (figure 3.9(a)) show that TCP New Reno without any modifications performs well for all the tested variations. Jitter causes a general increase in

[8] By "currently available", we mean variations that are available in the Linux kernel, and possible to compare by experimentation.

[9] Details on the performance of the different TCP variations for both thin and greedy streams can be found in the thesis of Espen Paaby [77].



(a) Successful 1st retransmission.

(b) Successful 2nd retransmissions.

Figure 3.9: Average retransmission delay, simplex streams, 100ms delay [50].

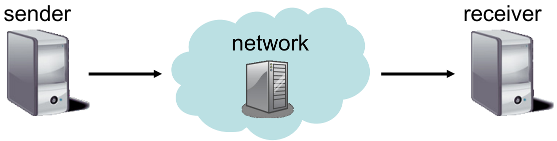
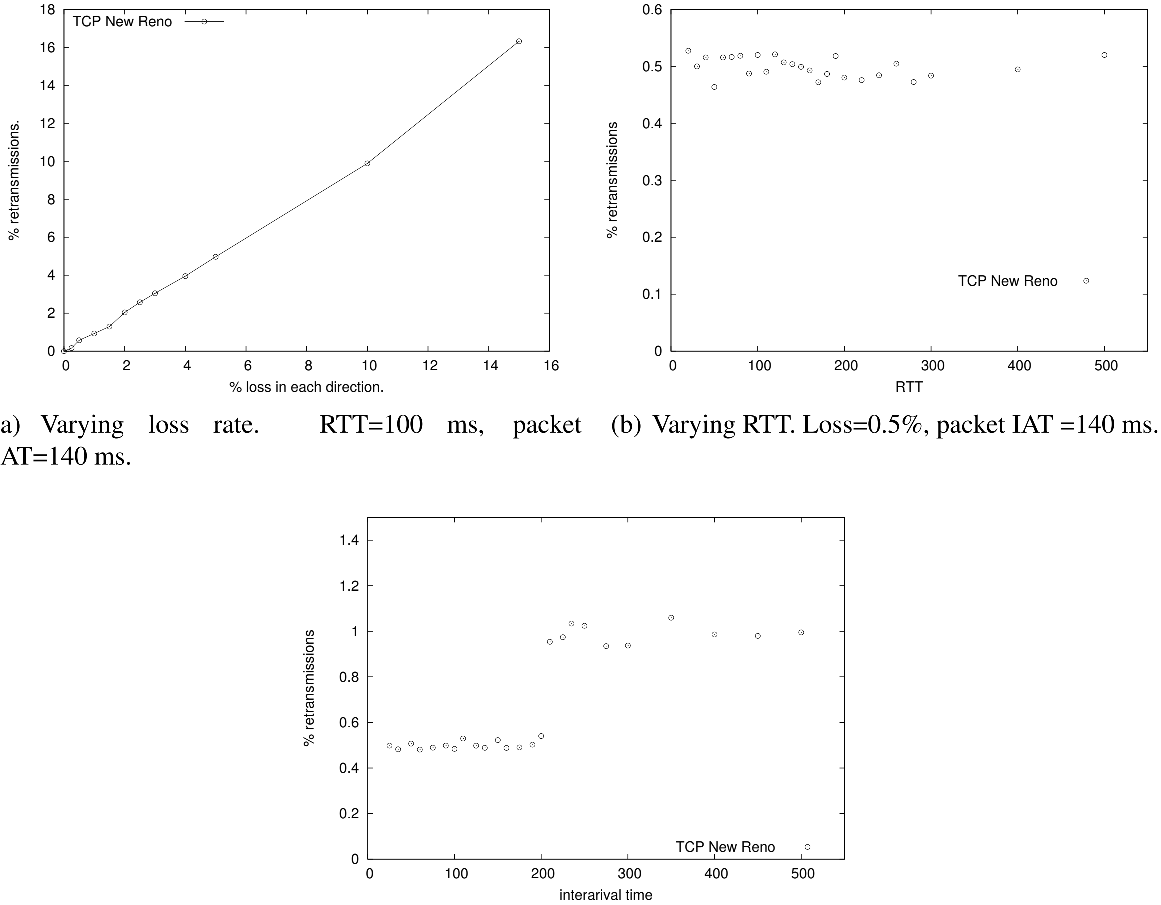


Figure 3.10: Test setup with an emulated network.

variance, while increased loss affects SACK, DSACK/FACK and Vegas the most. Figure 3.9(b) displays the statistics for 2nd retransmission. Here, the difference between variations is more pronounced, but New Reno still stands out as stable for all test parameters.

Testing TCP variations with thin-streams showed that TCP New Reno had the best overall latency-performance. To get a deeper understanding of how the thin-stream properties affect the number of retransmissions, we performed experiments where we varied basic properties (loss rate, RTT and packet IAT) and observed how this affected the number of retransmissions. Figure 3.11 shows results from the retransmission analysis. As expected, figure 3.11(a) shows a linear relationship between loss rate and retransmission rate. A higher loss rate increases the probability that more than one retransmission is needed to recover a lost segment. This inhibits a greedy stream, but severely degrade the latency-performance for a thin stream. The relative share of retransmissions needed to recover is independent of the connection RTT, as we can see from figure 3.11(b). Since the stream is thin, the majority of retransmissions are due to timeouts. The RTO value varies depending on the measured RTT and the RTT variance. If the RTT stays stable, the retransmission frequency reflects the loss rate as shown in figure 3.11(a). Figure 3.11(c) shows a scenario where we vary the IAT. We see an increase in the share of retransmissions as the packet IAT crosses 200 ms. This is caused by a combination of lost ACKs and the RTO that is calculated to $\sim (200ms + RTT) = 300ms$. When a segment is received, but the ACK is lost, the ACK from the next transmitted segment acknowledges the previous segment cumulatively. When the packet IAT crosses the 200 ms boundary, the next ACK fails to arrive before an RTO is triggered, and the segment is retransmitted.

The analysis of TCP variations showed that only small differences in the retransmission delays can be observed between the tested TCP variations. TCP New Reno provides the lowest overall latency for thin streams. Analysis of the retransmission properties indicate that high packet IATs can increase the chance of spurious retransmissions. Aside from that, there seems to be a near-linear relationship between loss rate and the number of retransmissions for thin streams.



(a) Varying loss rate. RTT=100 ms, packet IAT=140 ms.

(b) Varying RTT. Loss=0.5%, packet IAT =140 ms.

(c) Varying packet IAT, loss=0.5%, RTT=100 ms.

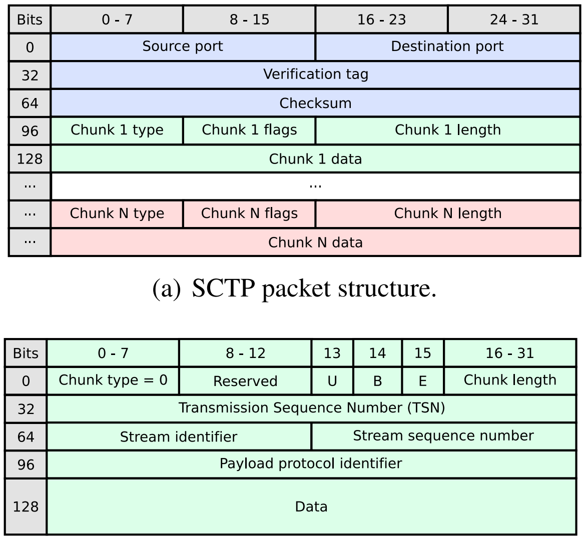
Figure 3.11: Analysis of changes in number of retransmissions for TCP New Reno when varying network and stream parameters. Packet size for all tests are 120 bytes [39].

3.3 SCTP

The stream control transport protocol (SCTP) [95] was originally designed for transporting Public Switched Telephone Network (PSTN) signalling traffic over Internet protocol (IP) networks by the IETF signalling transport (SIGTRAN) working group [93]. SCTP supports a range of functions that is critical to message-oriented signalling transport, but also has features and options that are useful for many other applications. After several rounds of modifications, SCTP has ambitions to become an ubiquitous protocol that ideally should be able to replace both TCP and UDP. This is enabled by the wide range of optional features that can be specified when setting up an SCTP connection[10].

SCTP is message- and connection oriented, meaning that one SCTP connection (often called an association) may serve several different data streams between the two hosts. Message bound-

[10] A factor that may contribute to the relatively slow adoption rate of SCTP is that the broad range of services makes the API for socket setup more complex. The overhead for connection setup and header-overhead due to separate chunk headers may also be inhibiting factors.



Bits	0 - 7	8 - 15	16 - 23	24 - 31
0	Source port		Destination port	
32	Verification tag			
64	Checksum			
96	Chunk 1 type	Chunk 1 flags	Chunk 1 length	
128	Chunk 1 data			
...	...			
...	Chunk N type	Chunk N flags	Chunk N length	
...	Chunk N data			

(a) SCTP packet structure.

Bits	0 - 7	8 - 12	13	14	15	16 - 31
0	Chunk type = 0	Reserved	U	B	E	Chunk length
32	Transmission Sequence Number (TSN)					
64	Stream identifier		Stream sequence number			
96	Payload protocol identifier					
128	Data					

(b) SCTP data chunk structure.

Figure 3.12: SCTP data packet structure.

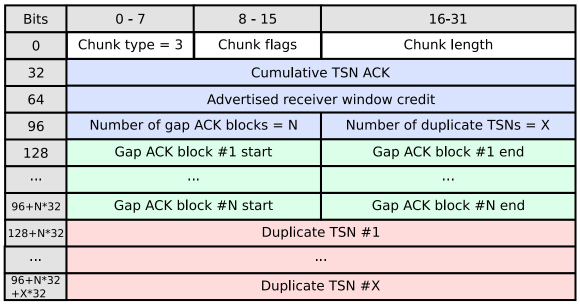
aries are preserved, in contrast to TCP which delivers a byte-stream to the receiver. Reliability is provided through acknowledged data delivery. Flow control is provided to prevent receiver overflow. The protocol also checks for bit errors and ensures that duplicates are removed. For improved fault tolerance, SCTP provides multihoming support, which allows for more than one IP address to be associated with the connection.

3.3.1 SCTP chunks and bundling

Preservation of message boundaries in SCTP is realised through message containers called "chunks". An SCTP packet consists of a generic SCTP header and a collection of chunks (see figure 3.12). Various tasks pertaining the connection are communicated through different chunk types (examples of chunk types are: DATA, INIT, SACK, ABORT and SHUTDOWN). Each chunk type has a different header structure reflecting the task of the chunk type.

Figure 3.12(b) shows the composition of the data chunk header. It contains the payload length, transmission sequence number (TSN), stream identifier, stream sequence number and payload protocol identifier. In addition, the header contains flags regarding fragmentation and ordering as well as a chunk type identification field. The TSN is a number that uniquely identifies the chunk in the SCTP connection, regardless of the stream that the chunk belongs to.

The separation of messages (chunks) from the packet data structure makes SCTP very flexible when it comes to the composition of each packet. Chunks do not have to be stacked in any special order. They do not have to originate from the same data-stream, and retransmissions can be stacked in between regular data chunks that are transmitted for the first time. The SCTP specification (RFC 4960 [95]) states that the user can request that SCTP bundle chunks when



Bits	0 - 7	8 - 15	16-31
0	Chunk type = 3	Chunk flags	Chunk length
32	Cumulative TSN ACK		
64	Advertised receiver window credit		
96	Number of gap ACK blocks = N		Number of duplicate TSNs = X
128	Gap ACK block #1 start		Gap ACK block #1 end
...	...		...
96+N*32	Gap ACK block #N start		Gap ACK block #N end
128+N*32	Duplicate TSN #1		
...	...		
96+N*32 +X*32	Duplicate TSN #X		

Figure 3.13: SCTP SACK chunk structure.

the path MTU allows. It is also stated that SCTP implementations may bundle chunks when congestion occurs, even if the user has not explicitly asked for it. Such “unsolicited bundling” may help improve delivery latency for thin-streams.

3.3.2 Acknowledgements in SCTP

For reliable transfer to work, acknowledgements have to be passed to the sender. For this purpose, SCTP applies a special chunk. Drawing on the experiences from the TCP developments, SACK-functionality is mandatory in SCTP. The SACK also incorporates functionality like the TCP duplicate SACK mechanism. The SACK chunk structure is shown in figure 3.13. The “cumulative TSN ACK”-field contains a TSN that acknowledges all transmitted chunks up to the cumACK TSN value. Following the cumACK are counters for the number of gap ACKs and duplicate TSNs are included in the SACK. At the end of the SACK chunk is the list of gap ACKs and, finally, the list of duplicate TSNs. The intrinsic inclusion of cumACK, gap ACKs and duplicate TSNs allows for SCTP to make qualified assumptions about chunk bundling and retransmissions.

SCTP also supports delayed SACKs (as described in section 3.1.3). An implementational choice is that delayed acknowledgements is enabled as default in the implementations that we have studied[11]. This can contribute to heightened retransmission delays.

3.3.3 SCTP RTO calculation

The RTO calculation specified in [95] is identical to the TCP RTO calculation for TCP described in section 3.1.2. In the same way as for TCP, this leaves the retransmission timer value vulnerable to RTT fluctuations (which yields overblown RTOs). There are differences in how it is commonly implemented, however. The recommended value for RTO_{min} is *one second* both for TCP and SCTP. While TCP implementations often soften this requirement, the SCTP im-

[11] We have used as reference FreeBSD 7.2 and Linux lksctp in the 2.6.28 kernel. There are more than 20 different SCTP implementations [112], though, and details may differ between them.

plementations we have studied adhere to the recommendation. The high RTO_{min} value helps to avoid spurious retransmissions, but can cause unwanted delays for interactive streams.

Another aspect of the SCTP RTO functionality is that the RTO timer is reset if a SACK arrives which acknowledges some, but not all, outstanding chunks. This is done to keep the *cwnd* open (by allowing dupSACKs to arrive so that a fast retransmit can be triggered) and to avoid spurious retransmissions. The extra delay that is added to the timer can, however, result in increased latency for thin streams. For streams with high packet IAT, this rarely (or never) happens. For interactive applications with a relatively low packet IAT (like Skype in table 2.1) over a high RTT connection, though, it can cause delays.

3.3.4 SCTP retransmission strategies

A salient difference from TCP (regarding the retransmission mechanisms) is that SCTP is message-oriented. Instead of retransmitting the previous packet(s) as TCP does, SCTP keeps track of the chunks that have timed out or been reported as lost and retransmits outstanding chunks. This makes the protocol more flexible with regard to packet composition and bundling. The SCTP implementation can apply bundling strategies to choose the optimal combination of chunks to be retransmitted with each packet. Chunks scheduled for retransmission can also be bundled with fresh data.

The fast retransmit mechanism (as described in section 3.1) is employed also by SCTP. After receiving three duplicate SACKs, a fast retransmission is triggered on the next outstanding chunk.

3.3.5 Other SCTP options

In addition to the basic SCTP features described in the previous sections, there are several options and extensions to SCTP that can prove valuable to different kinds of applications. One such extension is *partial reliability* [96], that can be used for time-dependent applications. This allows for optional reliability that enables UDP-like behaviour. This means that certain chunks can be tagged as “unreliable”, and are transmitted without the restrictions imposed by reliability and in-order delivery[12]. The option of *timed reliability* is also specified in [96]. *Timed reliability* makes it possible to invalidate a message in the sender buffer if a given timer has expired. This can save system and network resources in cases where the chunk should be dropped rather than be sent if the time limit is exceeded.

Another option that SCTP supports is *multi homing*. A *multi-homed* association keeps several IP addresses at one (or both) endpoints. This provides application-transparent robustness

[12] The partial reliability extension actually allows for any combination of reliability and in-order delivery (for instance unreliable, ordered delivery).

RTT (ms)	Type	Occurrences	Share	Min (ms)	Max (ms)	Avg (ms)
0	Retransmission timeout	282	76.2 %	999.1	1256.6	1005.5
	Fast retransmit	24	6.5 %	1024.4	1280.4	1088.4
	Reported lost and bundled	34	9.2 %	464.0	744.0	592.7
	Unsolicited bundling	30	8.1 %	231.8	744.0	274.7
100	Retransmission timeout	275	43.0 %	1039.9	1612.1	1049.8
	Fast retransmit	23	3.6 %	1126.5	1386.2	1173.1
	Reported lost and bundled	27	4.2 %	460.0	1356.1	689.3
	Unsolicited bundling	314	49.1 %	15.3	532.0	51.2
200	Retransmission timeout	266	40.1 %	996.2	1460.1	1144.6
	Fast retransmit	35	5.3 %	1228.4	1740.7	1274.2
	Reported lost and bundled	24	3.6 %	487.9	976.0	780.7
	Unsolicited bundling	338	51.0 %	28.0	888.0	172.8
400	Retransmission timeout	242	27.9 %	1343.0	1660.1	1352.0
	Fast retransmit	31	3.6 %	1427.2	1943.6	1496.2
	Reported lost and bundled	26	3.0 %	780.0	1430.1	1011.1
	Unsolicited bundling	567	65.5 %	11.8	832.0	213.4

Table 3.1: SCTP Retransmission statistics for thin streams, first retransmission [84].

against faulty network paths.

3.4 Evaluation of SCTP retransmission delays

The SCTP transport protocol was originally designed to handle signalling traffic. Such signalling traffic shows typical thin-stream properties. The flexibility of SCTP as a message-oriented protocol with options for partial reliability also made it interesting to test with thin-stream traffic. Thorough testing of SCTP properties[13] and comparison between SCTP and TCP was performed in [83]. The experiments analysed SCTP retransmission delays, both for greedy and thin streams. SCTP occasionally bundles unacknowledged chunks on retransmissions even though the chunk is not confirmed lost. Chunks that are registered as lost can be bundled with regular transmissions (of new data). In addition, retransmissions by timeout and fast retransmit are performed. Statistics were made of the frequency of each type of retransmission under different conditions. Tests were also made to compare the SCTP results to TCP New Reno for the same scenarios.

Table 3.1 shows the share of retransmissions caused by each of the retransmission methods SCTP uses. The delay before the sent data is cumulatively ACKed is also shown. The thin stream sent over the test network had a packet IAT of 250 ms and a packet size of 100 bytes.

[13] The SCTP version tested was the Linux kernel SCTP (lksctp) [13] available in the 2.6.16 Linux kernel.

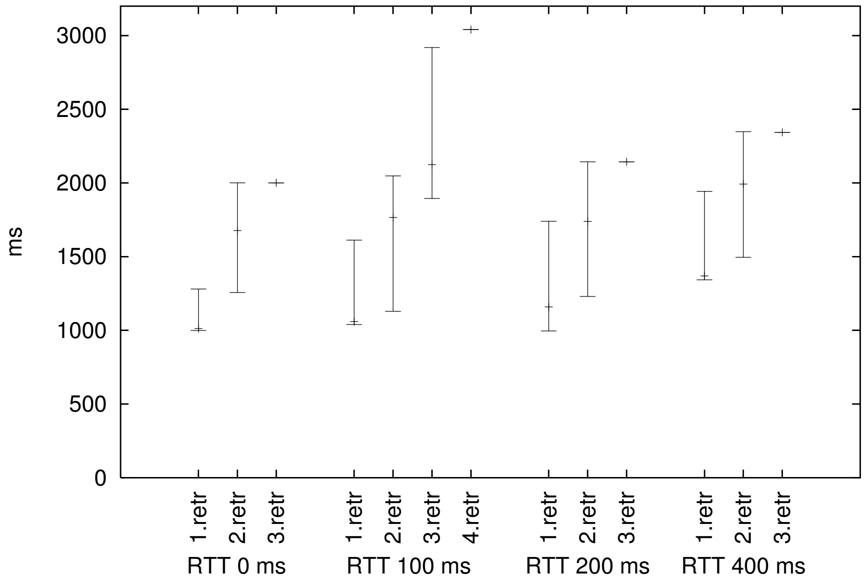


Figure 3.14: SCTP retransmissions by timeout [84].

The loss over the emulated network was 5% for these tests. The most prominent change as the RTT increases is that the number of unsolicited bundles increases drastically. From further analysis, it is clear that most of the unsolicited bundles represent spurious retransmissions. If this effect were to happen for semi-greedy streams, it would be unfortunate, as it would increase the bandwidth used drastically. A full-fledged greedy stream, however, would always fill the MSS, thus voiding any chance for unsolicited bundling. For thin streams, however, unsolicited bundling may be warranted if a faster recovery is made. As the RTT increases, (and unsolicited bundles also increase), the retransmissions by timeout also decrease. As retransmission by timeout is the mechanism that usually produces the largest retransmission delays, this effect is good with regard to latency. The fact that higher retransmission delays are seen for fast retransmit than for retransmissions by timeout can be explained by timer restarts[14].

Statistics for retransmissions by timeout are displayed in figure 3.14. We can see that the average delays before a retransmitted chunk is ACKed increases for each additional retransmission. The increase is, however, not exponential as would be expected from TCP. This is because SCTP can restart the timer when a late SACK arrives. The RTO can then be recalculated, and the backoff is reduced to a non-exponential mode. The fact that recovery does not occur until a second has passed, however, makes the delays bad for interactive applications.

SCTP was also compared to TCP New Reno, both with thin and greedy streams. Results from the tests with greedy streams showed that the fast retransmit-mechanism in SCTP yielded

[14] When the RTT is high, the chance increases for a special scenario where a chunk is first fast retransmitted, lost again, and retransmitted by a delayed timeout. The delay happens when incoming dupSACKs trigger a timer reset like described in section 3.3.3.

Loss Scenario	Extension	Retransmission Delay Statistics			
		Min	Max	Avg	Std
RTT = 0 ms	Plain	203.6	1632.1	231.7	98.6
	S	203.2	816.1	224.5	72.0
	S+D	202.9	1632.1	233.4	101.9
	S+D+F	200.1	1632.1	234.6	108.9
	S+F	200.1	1632.1	225.2	87.8
RTT = 100 ms	Plain	308.1	1216.1	328.3	97.7
	S	308.1	1264.1	348.5	113.9
	S+D	308.1	11185.2	388.4	554.4
	S+D+F	308.1	9816.6	360.3	378.4
	S+F	308.1	16901.0	392.6	708.7
RTT = 200 ms	Plain	412.1	6614.4	481.6	305.1
	S	412.1	3328.2	488.2	277.7
	S+D	412.1	3360.2	461.1	180.5
	S+D+F	412.1	2752.1	464.6	179.0
	S+F	412.1	5912.4	487.3	404.5
RTT = 400 ms	Plain	612.1	4960.3	728.4	437.2
	S	612.1	2842.5	692.5	264.2
	S+D	612.1	2480.2	693.0	239.4
	S+D+F	612.1	2480.2	708.7	286.2
	S+F	612.1	2480.2	697.8	246.1

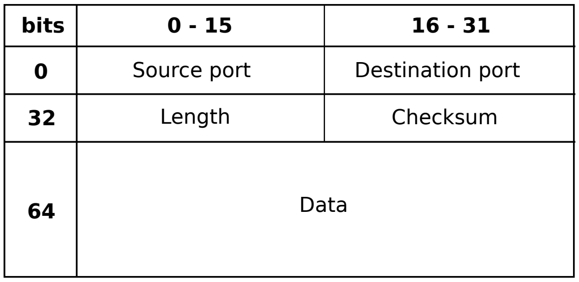
Table 3.2: TCP New Reno retransmission delays [84].

lower retransmission delays compared to TCP New Reno. The main contributing reason for this was that SCTP had no limit for the number of times a segment can be fast retransmitted before a cumulative ACK is received.

Table 3.2 shows the thin-stream retransmission statistics for tests performed on TCP New Reno. Packet IAT and size is the same as for the SCTP test described above. The abbreviations in the tables represents the following TCP mechanisms: *Plain* is standard TCP New Reno, *S* is the SACK extension, *D* is DSACK and *F* is FACK. Tests performed with thin-stream transmission patterns confirmed the TCP observations described in [77]. Average retransmission delays for SCTP and thin streams were actually higher than for TCP New Reno. Although surprising, considering that SCTP was designed for signalling traffic, this can be explained by three main contributing factors:

1. lksctp in the 2.6.16 kernel uses delayed SACKs as default, with no option to disable this feature. For greedy streams, this limits the upstream traffic without any significant performance penalty. For interactive thin streams, delayed SACKs increase the time before the sender is able to react to loss.

2. The RTO_{min} of the tested SCTP implementation was set to 1 second (in contrast to TCP



bits	0 - 15	16 - 31
0	Source port	Destination port
32	Length	Checksum
64	Data	

Figure 3.15: UDP header structure.

New Reno's RTO$_{min}$ of 200 ms). This reduces the number of spurious retransmissions for greedy streams, but increases the retransmission delay for thin streams unable to fast retransmit.

3. At the time when our tests were performed, the *lksctp* implementation needed 4 duplicate SACKs to trigger a fast retransmission[15]. This increases the delay before fast retransmitting, and makes SCTP respond more slowly than TCP in the cases where fast retransmissions are possible.

The high RTO$_{min}$ in SCTP means that if the data segment is recovered after only one retransmission, TCP New Reno is preferred. If several retransmissions of the same segment are needed, the SCTP property of multiple fast retransmissions before backing off yields lower retransmission delays for SCTP than for TCP New Reno. SCTP bundling of unacknowledged segments can compensate for some of the increased retransmission delay, but latencies are still too high to satisfy interactive applications when loss occurs.

3.5 UDP with application layer reliability

The simplest protocol among the transport protocols is the User Datagram Protocol (UDP) [90]. It provides addressing (port number) and error checking (checksum). Messages are sent as they are delivered from the application layer, giving the sender control of the transmission timing. No correction is done from the protocol if packets are lost, duplicated or reordered. The communication setup demands no handshake, and the transmission logic is stateless. UDP can support multicast and broadcast.

Figure 3.15 shows the UDP header. Source and destination port numbers enable application multiplexing. Checksumming makes error detection possible. The simplicity of the UDP header makes the payload to header ratio large. UDP gives the application designer freedom to

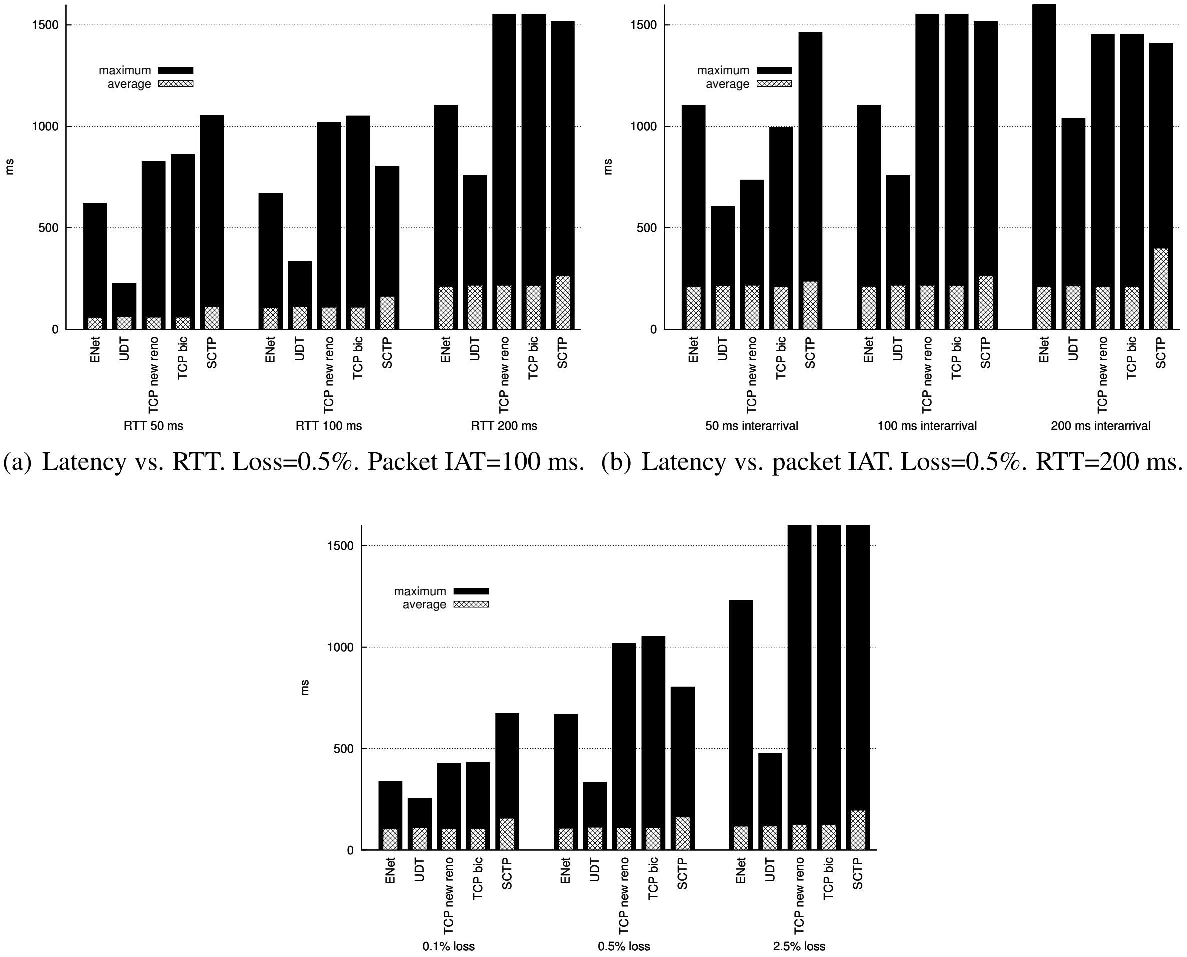
[15] The use of 4 dupSACKs was according to the first SCTP RFC [97], published in year 2000. When the revised RFC for SCTP [95] was published in 2007, the number of SACKs needed to trigger a fast retransmission was changed to 3 (as used in TCP).

control the transmissions, but leaves much to be wanted with regard to security. The absence of reliability and congestion control makes UDP streams likely to cause congestion when used for data-intensive applications. Since application designers must implement services that are needed (like support for reliability and sequenced delivery) on the application layer, implementations show great variation in fairness.

UDP is preferred to TCP for many applications with special timing needs (like media streaming and fast-paced online games). To provide services needed by the applications (like reliability and in-order delivery), frameworks must be developed that can be configured to suit the developer's needs. Such network libraries and middleware come in many variants and levels of abstraction. Some examples include ACE [3], ENet [37], HawkNL [7], Plib [12], SDL [11], ClanLib [5], Net-Z [8], RakeNet [9], ReplicaNet [10], UDT [52, 14] and ZoidCom [15]. Two main strategies are common for such application-layer frameworks: 1) a low-level network library with basic services, and 2) a more complex library with many options and a high level of abstraction. We next present one example of each type of library.

UDT [52] provides various features based on UDP transport. A structured (socket-like) interface provides a high level of abstraction making it easy for application developers to use. Partial reliability and in-order delivery are supported, and congestion control mechanisms enable UDT to maintain a level of fairness. A wide range of parameters can also be set, in order to achieve the combination of options that is best for each application. Moreover, UDT divides its packets into control and data messages. In order to keep track of the status of the remote host, keep-alive messages are an integral part of the framework. This combined with aggressive bundling strategies, used if less than the estimated bandwidth is consumed, contributes to a large redundancy rate for the UDT platform. Retransmissions are managed as for TCP with timeouts and dupACKs, and additionally using negative acknowledgements.

ENet [37] aims for online gaming support. It was developed for the Cube game engine [35] and was later used by other networked games. ENet provides a relatively thin, simple and robust network communication layer on top of UDP. It provides optional, reliable, in-order delivery of packets. ENet is a small library that provides some functionality without supplying a high level of abstraction and can therefore not be considered a middleware platform. The services include a connection interface for communicating with the foreign host. Delivery can be configured to be stream oriented or message oriented. The state of the connection is monitored by pinging the target, and network conditions such as RTT and packet loss are recorded. Partial reliability is supported, and retransmissions are triggered using timeouts based on the RTT, much like the TCP mechanisms. The congestion control implements exponential backoff, making it vulnerable to bursts of loss, and ENet also applies bundling of queued data if the maximum packet size is not reached.



(a) Latency vs. RTT. Loss=0.5%. Packet IAT=100 ms. (b) Latency vs. packet IAT. Loss=0.5%. RTT=200 ms.

(c) Latency vs. loss rate. RTT=100 ms. IAT=100 ms.

Figure 3.16: Perceived application layer latency.

3.6 Analysis of retransmission delays for UDP and application layer reliability

For very latency-sensitive applications, like FPS games, UDP is the most common choice of protocol. In the cases where reliability and in-order delivery must be provided, this is implemented on the application layer. We wanted to test the performance of a couple of frameworks that provide such service against TCP and SCTP in order to learn more about latency performance. We chose two different frameworks: 1) ENet, that is designed explicitly for game traffic, and 2) UDT which tries to use “free” bandwidth to avoid retransmissions.

Figure 3.16 shows results from tests where thin streams were transmitted over an emulated network. We varied the RTT, packet IAT and loss rate to see how this would influence the delivery latency. There are only very small differences in the average delivery latency for all the tests. SCTP has a higher average latency because of the high RTO_{min} value in the lksctp implementation. The timer reset mechanism in SCTP can also contribute to its high average delay, since this function negatively effects the latency for thin streams. When we study the maximum delivery latency, we see large differences between the tested alternatives. In figure 3.16(a), the RTT is varied, while IAT and loss is kept constant. We can see that UDT has the lowest maximum delay for all RTTs. ENet also performs better than the TCP and SCTP variations. The reason for this is probably that UDT bundles aggressively, and estimates the "free" bandwidth to be large, especially for low RTTs. ENet has more aggressive retransmission techniques since it does not implement congestion control.

When the IAT is varied (figure 3.16(b)), UDT still shows the lowest maximum latency overall. For the 50 ms IAT test, however, TCP New Reno performs nearly as well as UDT. In the 200 ms IAT test, all tested alternatives have a maximum latency of above one second. In this case, ENet shows the highest delivery latency. For the 200 ms IAT test, we can also note that the average latency for SCTP is almost double of the other tested alternatives.

As expected, a low loss rate keeps the maximum latency down for all the tested alternatives. Figure 3.16(c) shows the results when the loss rate is varied. The lowest maximum delivery latency is still provided by UDT. As the loss rate is increased to 2.5%, all the tested alternatives (except UDT) are well above one second in maximum latency.

The overall impression is that UDP with the tested application layer frameworks is able to provide lower maximum latency. This comes at the cost of redundancy and the lack of congestion control. There is also the consideration that UDP is often blocked by ISP firewalls. Another observation from the test results is that high IATs seem to make the maximum latency very high for all the tested alternatives. Thin streams often have very high packet IAT, so this should be considered also when implementing frameworks for reliability and in-order delivery on top of UDP.

3.7 DCCP

Datagram Congestion Control Protocol (DCCP) [63] provides congestion control without providing reliability. This is beneficial when timed delivery is desirable. DCCP is connection-oriented, providing reliable setup and teardown. Flow-control is provided, and the setup enables negotiation of optional features. Applications that need to control delivery timing can do so without having to implement congestion control. This provides a level of flexibility in between TCP and UDP.

While DCCP provides a combination of services that is useful for many classes of applications, the fact that it has to be commonly supported in different operating systems is still a limitation. As long as TCP provides reliable services and UDP can be extended on the application layer, most developers still keep to combinations of the two.

For this work, DCCP has not been explored since TCP, UDP and SCTP all are more widely available, and they can provide the same set of services. The congestion control feature of DCCP also has little influence on the high-IAT thin-stream scenario, and can therefore readily be investigated using the more common protocols.

3.8 Related transport protocol mechanisms

Other projects have performed work in the field of time-dependent delivery over reliable protocols. Some of this work is also relevant for the thin-stream scenario. This section discusses approaches that are relevant to this thesis in one or more aspects.

3.8.1 Timer calculation

Ekström and Ludwig [36] point out that the retransmission timeout algorithm defined in RFC 2988 [82] and used in both TCP and SCTP responds sluggishly to sudden fluctuations in the RTT. This leads to extreme estimated RTO values in some cases. They also point out that the RTTVAR computation does not distinguish between positive and negative variations, and therefore increases the RTO in the case of both RTT increases and decreases. Their proposed algorithm alleviates the consequences of RTT fluctuations and is, as such, a good addition to the main protocol for a range of special cases. Their findings are consistent with our observations made in [84] of high RTO values that are worsened by the SCTP delayed acknowledgement algorithm. While their solution leads to a more stable RTO, it is on average higher than that proposed in RFC2988, which is not desirable for our scenario.

3.8.2 Exponential backoff

Ekström and Ludwig also mention that a less conservative exponential backoff algorithm [36] should be considered, which is one of the mechanisms that we investigated. The removal of exponential backoff for special cases when using SCTP for telephony is suggested by RFC 4166 [34]. The document warns, however, about the effect this may have on congestion. The general removal of exponential backoff in order to recover more quickly after loss events is a controversial issue that has been discussed also by Mondal and Kuzmanovic. They argue that exponential backoff is not necessary to maintain the stability of the Internet [75]. As this change will, possibly, lead to longer recovery-time after congestion events as streams retransmit more aggressively, care should be taken to introduce such changes on a general basis. Thin-stream applications, however, put very little pressure on a bottleneck due to the low packet IATs of the stream. They therefore contribute very little to congestion events, and do not expand their congestion window as the application limits the window size.

3.8.3 Fast retransmit modifications

The problem of late retransmissions has been addressed by the optional Early Fast Retransmit (EFR) mechanism[16] which exists in FreeBSD for SCTP and has been used for tests and comparisons in this thesis. The EFR mechanism is active whenever the congestion window is larger than the number of unacknowledged packets and when there are packets to send. It starts a timer that closely follows $RTT + RTTVAR$ for every outgoing packet. When the timer is triggered and the stream is still not using the entire congestion window, it retransmits all packets that could have been acknowledged in the meantime. An EFR timeout does not trigger slow start like a normal timeout, but it reduces the congestion window by one.

In an IETF draft, Allman et al.[17] suggested that measures should be taken to recover lost segments when there are too few unacknowledged packets to trigger a fast retransmission. They proposed Early Retransmit (ER), which should reduce waiting times in four situations: 1) the congestion window is still initially small, 2) it is small because of heavy loss, 3) flow control limits the send window size, or 4) the application has no data to send. The draft proposed to act as follows whenever the number of outstanding segments is smaller than 4: if new data is available, it follows Limited Transmit [16], if not, it reduces the number of duplicate packets necessary to trigger fast retransmit to as low as 1 depending on the number of unacknowledged segments. It differs from our approach in two ways. The first is the motivation (our goal is to reduce latency for consistently thin streams). The second is that Allman et al. try to prevent retransmission timeouts by retransmitting more aggressively, thus keeping the congestion window open even though congestion may be the limiting factor. If their limiting conditions change, they still have higher sending rates available. Our applications are not limited by congestion control. We have no motivation to prevent retransmission timeouts in order to keep the congestion window open because in the thin-stream scenario, we do not need a larger window, but we retransmit early only to reduce application-layer latencies. We are therefore combining the approach with a reduced minimum retransmission timeout (RTO$_{min}$) to handle the worst-case situations instead of preventing the retransmission timer from firing. ER is less frequently active than EFR, but it is more aggressive when the number of unacknowledged packets is small.

Hurtig and Brunström suggested a modification to the ER algorithm from working on a *number of bytes*-basis as originally proposed, to a *number of packets*-basis [57]. This increases the effectiveness of the ER algorithm for signalling (and thin-stream) scenarios.

Brennan and Curran [26] performed a simulation study for greedy traffic and identified weaknesses in the fast retransmit procedure. They propose to modify the fast retransmit mech-

[16] This mechanism can be enabled in FreeBSD by using the $net.inet.sctp.early\_fast\_retran$ syscontrol. We have, however, not been able to find any published papers which yields further details of the mechanism's implementation in FreeBSD.

[17] IETF Draft draft-allman-tcp-early-rexmt-05: Mark Allman, Konstantin Avrachenkov, Urtzi Ayesta, Josh Blanton, "Early Retransmit for TCP and SCTP", June 2007, expired Dec. 2007.

anism to allow a given chunk to be fast retransmitted only once if no timeouts intervene. This would make the fast retransmission scheme more conservative, and reduce the chance of poor network utilisation due to prolonged congestion events. Their modifications would, however, increase delays for thin streams.

3.8.4 RTT estimation and congestion detection

The removal of the exponential back-off can of course result in spurious retransmissions when the RTT changes. The proposed method of TCP Santa Cruz [81] uses TCP timestamps and TCP options to determine the copy of a segment that an acknowledgement belongs to and can therefore provide a better RTT estimate. Since the RTT estimate can distinguish multiple packet losses and sudden increases in actual RTT, TCP Santa Cruz can avoid exponential back-off. The ability of TCP Santa Cruz to consider every ACK in RTT estimation has minor effects in our scenario where hardly any packets are generated. The ability to discover the copy of a packet that an ACK refers to would still be desirable but would require receiver-side changes that we avoid.

3.8.5 RTO$_{min}$ and delayed SACKs

Grinnemo and Brunström [49] discuss the problem of RTO$_{min}$, and propose a reduction to fulfil the requirements of RFC 4166 [34], an RFC on the applicability of SCTP for telephony. The RFC itself discusses problems and solution approaches, and it proposes to choose the path within a multi-homed association that experiences the shortest delay, an approach that may be used as a supplement to other techniques for thin-stream scenarios. The RFC considers reduction of the RTO$_{min}$, but warns that this solution may have negative effects on network behaviour. Removing delayed SACK is mentioned without stating any side-effects. This would also be beneficial in our scenario. However, it is a receiver-side change, while we aim exclusively at sender-side changes. Of the discussed options, we choose the removal of the exponential back-off, but instead of doing it arbitrarily, we limit it to situations where fast retransmit is impossible due to lack of unacknowledged packets (i.e., too few packets in flight).

3.8.6 Unreliable and partially reliable transport

Problems with carrying time-sensitive data over SCTP were presented by Basto and Freitas [20]. The traffic that they considered was loss-tolerant, and they proposed the use of SCTP's partial reliability extensions [96]. Ladha et al. [65] examined several methods of detecting spurious retransmissions and proposed modifications that would increase throughput but also increase the latency of individual lost packets.

Lundqvist and Karlsson presents a solution for TCP with integrated forward error correction (FEC) in a paper from 2004 [68]. Such a solution helps to reduce retransmission latencies for thin streams by avoiding the retransmissions themselves. This solution will, however constantly increase bandwidth for the stream, especially in greedy-stream scenarios. For thin-stream scenarios, the effect is limited because of the need for sent data in order to have an effective FEC. It does also require modifications to both sender and receiver, and is therefore not one of the principles we consider in relation to our sender-side only modifications.

3.8.7 Latency-sensitive streams

The earlier work that has been done in the field of reducing latency upon retransmissions all focus on special cases of greedy streams [64, 46, 108, 45, 109] where measures can be taken to improve performance of the target applications. Our work identifies thin streams as time-critical and latency sensitive. We therefore apply a combination of several modifications upon detection of the thin stream, and can thus improve latency for the stream in a manner not yet explored in literature.

3.9 Thin-stream challenges

In chapter 2, thin-stream properties and latency requirements are discussed. Experiments presented in sections 5.2 and 3.4 show how thin-streams suffer from high latencies when using reliable transport protocols. When UDP-based application layer techniques were evaluated in section 3.6, the tested frameworks showed high maximum delays for high packet IAT (thin) streams. A wide range of different alternatives were evaluated, all producing the unwanted retransmission latencies that were pronounced in the Anarchy Online traces (shown in section 2.4). In order to explain the behaviour of common retransmission mechanisms, we have reviewed the key concepts of TCP and SCTP congestion control and reliability mechanisms. Implementations of reliability and in-order delivery on top of UDP is also very often modelled on the principles from TCP.

The foremost tool used by TCP to recover without triggering a timeout is the *fast retransmit*-mechanism. This is also the key to understanding the high latencies that can be observed for thin streams. When the stream is thin, the congestion window never expands to be limited by congestion control. Thin streams often have no more than one packet in flight per RTT. As a fast retransmit needs three dupACKS to be triggered, this seldom (or never) happens for such streams. The effect is that recovery for thin streams is limited almost entirely to timeouts. A retransmission by timeout triggers exponential backoff, thus delaying further retransmission attempts. Subsequent lost retransmissions increases the delay until we can observe extreme

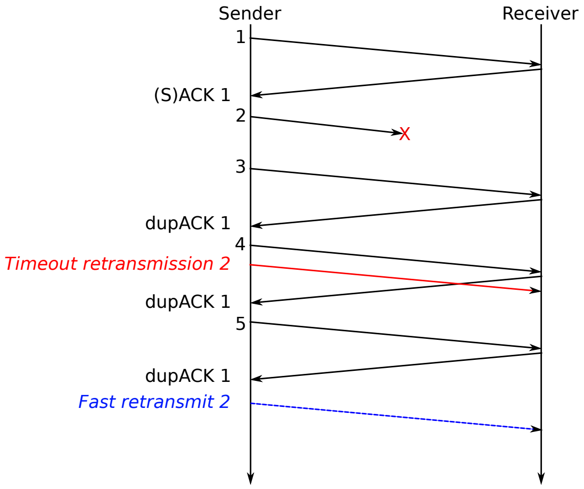


Figure 3.17: Fast retransmit with thin streams.

values (like the ∼67 second delay observed in the Anarchy-online trace from section 2.4 after 6 retransmissions of the same packet).

Figure 3.17 shows an example of a transmission pattern for a thin stream experiencing loss. In the example, the application produces less than one packet per RTT, meaning that an ACK is received before the next transmission (if no loss occurs). When a packet is lost, one dupACK may be received before a timeout is triggered. As three timeouts are needed to trigger a fast retransmission, timeouts dominate retransmissions. If the same segment is lost several times, exponential backoff soon raises delivery latency to a scale of several seconds. As the thin stream often keeps its transmission properties (packet IAT) throughout its lifetime, the effect that all retransmissions happen by timeout delays recovery every time loss occurs, influencing the experience of users running thin-stream applications like games, VoIP, remote desktops and shells. Thus, for interactive applications, which very often display thin-stream properties, such high delays cause latency-events that degrade application performance and reduce the QoE. We can therefore conclude that support for such interactive applications should be improved. As a basis for investigations into such improvements, all mechanisms that affect application layer delay for reliable protocols should be taken into consideration. As greedy streams perform well as regards latency, we want to separate greedy and thin streams, and apply new techniques to improve latency only when warranted. In the next chapter, we present modifications that we have developed for TCP and SCTP with the aim of improving retransmission delays for thin streams.

Chapter 4

Thin-stream modifications

We have seen from the analysis of TCP and SCTP performance that the largest contributing factor to thin-stream latency is the retransmission strategy. Retransmission timers, exponential backoff and the fast retransmit algorithm do not work optimally for thin streams. In this chapter, the modifications that we developed to improve the retransmission latency are described. When considering how to improve the latency situation for thin streams, we formulated the following prioritisation:

- The underlying reason for the high latencies we observed in thin-stream scenarios over reliable protocols was the inability to trigger fast retransmissions. Hence, we want to investigate and modify mechanisms related to fast retransmit.

- Since thin streams never probe for bandwidth, the change in bandwidth usage when it enters recovery (withdraws) is minimal. When the thin stream goes into exponential backoff, it results in very high latencies with insignificant effects on congestion. For that reason, we want to investigate and modify the exponential backoff mechanism.

- The analysis of thin-stream applications revealed that the payload is generally very small. Based on this observation, we wanted to investigate ways to utilise "free" space in the sent packets (when packet sizes are significantly smaller then the maximum transmission unit (MTU) size) to improve the delivery latency when loss occurs.

- Other protocols have been devised that give better support for timely delivery while providing the same services as TCP (or SCTP). New protocols, however, need to be supported on both sides of the connection. We wanted to create transparent, sender-side, standards compliant modifications, so that any unmodified receiver may benefit from a modified sender.

- TCP-based congestion control- and recovery schemes are well-tested and well-functioning for greedy streams, and changing the general behaviour of TCP and SCTP would be counterproductive (and unwise). Therefore, we wanted to separate the thin streams from the greedy using a metric, and apply modifications to reduce latency *only* when the system identifies the stream as thin.

Working from these premises, we developed a set of modifications for TCP and SCTP. The following sections describe the rationale behind, and the changes made, to TCP and SCTP in the Linux kernel.

4.1 TCP modifications and implementation

Our proposed set of mechanisms consists of a way to identify the thin streams, and then change the retransmission mechanisms to adapt to thin-stream behaviour. In addition, we have developed and tested a bundling mechanism that takes advantage of the small packet sizes that thin-stream applications very often generate. By bundling unacknowledged segments with new data, we aim to reduce the delivery latency. Our proposed modifications and their implementation in the Linux kernel are presented in the following sections.

4.1.1 TCP thin stream detection

In order to separate greedy streams from thin, a suitable metric (or set of metrics) for distinction had to be chosen. A wide range of metrics and modifiers can be used to this end. The thin-stream detection mechanism must be able to dynamically detect the current properties of a stream, and separate greedy streams from thin. Since the process needs to be continuous, effectiveness should be a criterion. This means that it should ideally be based on variables that are already available in the TCP code. The identification mechanism should be able to identify the thin streams as precisely as possible, with eventual false positives reduced to a minimum (or none). The application should not have to be aware of the detection mechanism, nor have to feed data about its transmission rate to the network stack; it should be transparent to the application.

Choices

Application of the thin-streams modifications must be dynamic in order to use regular TCP mechanisms when the stream is not thin. This means that a wrapper has to be applied each time an affected retransmission mechanism is called. Figure 4.1 shows how each modification is encapsulated in code that selects between mechanisms based on the current stream properties. The key to a proper identification is the *tcp_stream_is_thin*-test. The primary identifying property for the thin streams, the high packet IAT, is also the main reason for the high retransmission delays we have observed. A natural path of investigation is therefore to find a way to gather

```
if ( tcp_stream_is_thin ) {
    apply modifications
} else {
    use normal TCP
}
```

Figure 4.1: Determining whether to use thin-stream modifications.

and use information about the packet IAT from the application. A system of fine-grained timers (as described for TCP Vegas in section 3.1.4) could help to accurately detect the relationships between link RTT and packet IAT. This would tell us whether the feedback was sufficient for the retransmissions to work satisfactorily, or if extra measures should be taken to reduce latency. This would, however, require extra calculations as well as fine-grained timer support. Another option would be to let the application report its transmission patterns to the transport layer. As mentioned, an exchange of information between the transport layer and the application layer would complicate the implementation and jeopardise transport-layer transparency. We therefore look to the existing transport-layer variables to identify a suitable mechanism for thin-stream detection. The mechanism we chose is based on the already existing counter of unacknowledged packets.

$$in\_transit \leq (ptt_{fr} + 1) \tag{4.1}$$

Equation 4.1 shows the criterion we implemented to classify a stream as thin and apply the enhancements. Here, *in_transit* is the number of packets in flight[1] and ptt_{fr} is the number of packets required to trigger a fast retransmission (3 for Linux 2.6.23.8). By adding one to the ptt_{fr} limit, we take into consideration that a packet has to be lost for a fast retransmission to be triggered (1 lost packet + 3 dupACKs = 4 *in_transit*). Experiments were performed where we also considered the connection RTT and the estimated loss rate to adjust the limit for triggering thin-stream mechanisms. Loss rate was taken into consideration for our SCTP-experiments.

The algorithm could be extended by taking into consideration the number of packets in flight over time, but that would require the introduction of new variables and more calculations. Other modifiers could also be applied, like the measured RTT and loss rate. A high RTT connection would yield high retransmission delays due to an inflated RTO. It would also leave room for more packets in flight. In such cases, the thin-stream requirement could be relaxed to compensate for the potentially raised latencies upon retransmissions. A high loss rate would also be bad for thin streams that would go into exponential backoff, and would not probe for bandwidth

[1] Packets in flight , or "*FLIGHT SIZE*" as it is called in the TCP specifications, are packets that are transmitted, but not yet acknowledged.

upon recovery. Therefore, a modifier for high loss rates could be implemented. We have, however chosen the most conservative approach for the TCP thin stream triggering mechanism: to keep it at a static level determined by the current *in_transit* variable and the transport protocol's ptt_{fr}. This conservative choice limits the activation of modifications to the cases where fast retransmit *cannot* be triggered. As equation 4.1 shows, the triggering mechanism relies only on counting transmitted but unacknowledged packets and uses neither packet send times nor additional SACK information to draw further conclusions. This is because the tests presented in section 5.2 showed no detectable latency improvement from such mechanisms.

Implementation

We created a new inline function in *include/net/tcp.h* to provide efficient checking of the current number of packets in flight. This function could then be used by all thin-stream mechanisms. Figure 4.2 shows the code that performs the check. In our implementation we operate with a "THIN_LIMIT" of 4, to reflect the number of dupACK needed to trigger a fast retransmit. The TCP socket struct (*tcp_sock*) is used to hold information necessary to manage a socket. The member *packets_out* is a counter of the updated number of packets in flight. By making this function inline, we can increase efficiency for this function that is called each time a related retransmission mechanism is to be triggered.

```
static inline unsigned int tcp_stream_is_thin(const struct tcp_sock *tp)
{
        return (tp->packets_out < THIN_LIMIT));
}
```

Figure 4.2: Test for determining whether there are less than 4 packets in flight, located in *include/net/tcp.h*.

4.1.2 Switches for enabling the thin-stream modifications

Even though the modifications are triggered dynamically based on whether the system currently identifies the stream as thin, the basic enabling of each mechanism should be optional to the user (system). By implementing switches to enable the modifications, we allow for application designers / server administrators to choose whether the modifications are needed for their system setup. For instance, in a scenario with no time-dependency, there would be no large rationale to take extra measures to avoid second-long delays. We have implemented two alternative ways of enabling each mechanism:

1. Linux system-wide *syscontrol*: A *syscontrol* is a variable that is set by a system administrator to control operating system properties on a system-wide basis. The variable is

Mechanism	sysctl	IOctl	Input
LT	tcp_force_thin_rm_expb	TCP_THIN_RM_EXPB	Bool
mFR	tcp_force_thin_dupack	TCP_THIN_DUPACK	Bool
RDB	tcp_force_thin_rdb	TCP_THIN_RDB	Bool
RDB bundle limit	tcp_rdb_max_bundle_bytes	N/A	Num

Table 4.1: Mechanisms implemented for triggering the modifications.

```
sysctl net.ipv4.tcp_force_thin_rm_expb=1
```

(a) Syscontrol enabling of modification.

```
echo “1” > /proc/sys/net/ipv4/tcp_force_thin_rm_expb
```

(b) /proc-variable enabling of modification.

Figure 4.3: Examples of how to enable the LT thin-stream modification.

defined in *sysctl_net_ipv4.c* and updated/kept in *include/net/tcp.h*.

2. Linux per-stream *IOcontrol*: When creating the socket, the application designer can choose to enable available mechanisms for this stream only. For TCP, the variable is kept in the *tcp_sock* struct (see section 4.1.1) and can be accessed for reference if the respective socket is used.

If the *syscontrol* setting is enabled, this enables the modification for all streams, regardless of the settings of corresponding *IOcontrols*. By allowing both methods for enabling the implemented mechanisms, it is possible to use the *syscontrol* to improve latency for proprietary thin-stream applications where the code cannot be inspected or modified. The use of *IOcontrols* allows for flexible enabling of each implemented mechanism based on the program designer's intimate knowledge of the application's needs and expected stream properties.

Table 4.1 shows a list of the *syscontrols* and *IOcontrols* that can be used to enable our implemented mechanisms. The "Input"-column shows the parameters given to the control. "Bool" signifies an on/off trigger, and "Num" means that a value can be specified. The modifications can be enabled system-wide through the syscontrol as shown in the examples presented in figure 4.3. An alternative way is to write the value to the *proc*-file system variable for this syscontrol as shown in figure 4.3(b). Root access is needed to set such system-wide parameters. An example of how to assign a limit for RDB bundling is shown in 4.4. IOcontrols are specified in the source code of the application that creates the socket. After the socket is created, the IOcontrols can be used to enable custom properties for the specific socket. Figure 4.5 shows an example of how the IOcontrol is enabled in Linux using the `setsockopt()` function. When this method is used for enabling the mechanisms, each socket can be assigned different combinations of the mechanisms for better customisation and adaptation to the intended use of

```
sysctl net.ipv4.tcp_rdb_max_bundle_bytes=600
```

Figure 4.4: Example of the use of a syscontrol to limit RDB bundling to 600 bytes.

```
        int flag = 1;
        int result = setsockopt(sock, IPPROTO_TCP, TCP_THIN_RM_EXPB,
                                (char *) &flag, sizeof(int));
```

Figure 4.5: Example of the use of a IOcontrol to enable the LT thin-stream modification.

the socket.

4.1.3 Linear retransmission time-outs

The most extreme latency-events that were observed in our thin-stream analysis occurred when the same data segment had to be retransmitted several times due to timeouts. This triggers exponential backoff which, depending on the RTO, soon reaches second-long delays. We therefore wanted to explore modifications to the use of exponential backoff for thin streams.

As the RTO calculation is one of the fundamental parts of TCP recovery and congestion control, we want to keep the existing mechanisms for RTO calculation. The exponential backoff, however, causes very large delays. As thin streams does not expand the congestion window, the exponential backoff mechanism tries to hold back a stream that is not aggressively probing for bandwidth. Our goal was therefore to implement a mechanism that uses linear timeouts based on the calculated RTO when the stream is identified as thin.

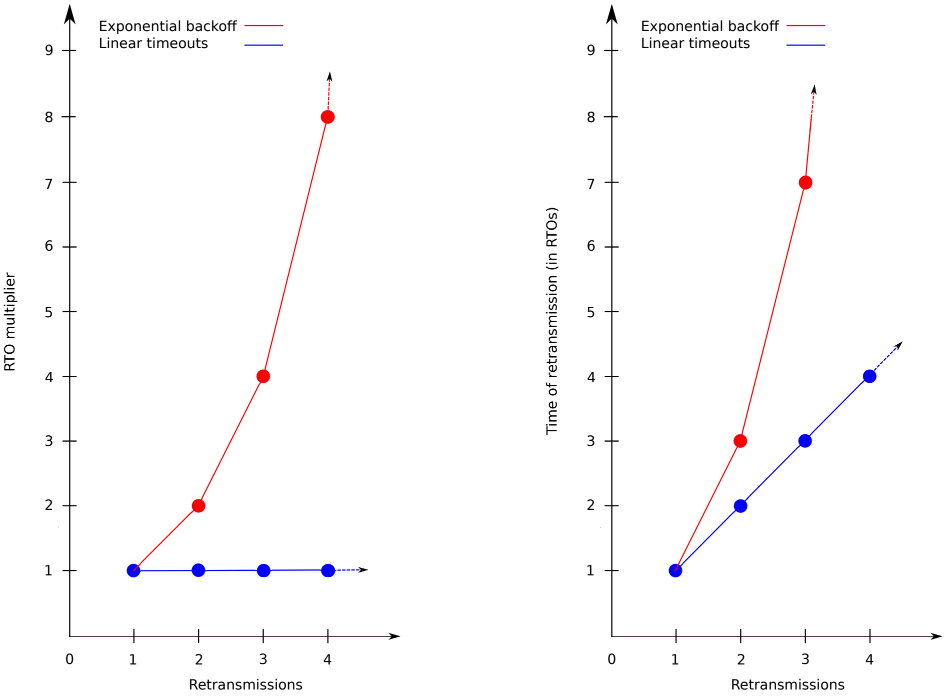
Choices

The modification we propose for this case of retransmission is to keep linear timeouts as long as the stream registers as thin (using the test described in section 4.1.1).

Figure 4.6 shows a comparison of RTO (see section 3.1.2) values for subsequent timeouts with linear and exponential backoff timers. Since the time it takes to react to feedback is limited by the RTT, a high-RTT connection suffers more from exponential backoffs than a low-RTT connection. For the first retransmission, exponential backoff is not in effect, but as the number of consecutive timeouts increases, the added retransmission delay soon gets very high[2].

A series of consecutive timeouts can be caused by loss of data, but can also happen when (S)ACKs are lost or delayed due to delayed (S)ACKs on the receiver. By retransmitting more

[2] Assuming that the initial RTO is 300 ms, the fourth retransmission of the same lost segment would happen after $300 + 300 * 2 + 300 * 4 + 300 * 8 = 4500$ ms.



(a) RTO multiplier

(b) Retransmission send time

Figure 4.6: Difference between linear timeouts and exponential backoff.

quickly when the congestion window is collapsed, we can avoid the largest retransmission delays both when data and ACKs are lost. As an addition, or alternative scheme, a change in the RTO calculation may be devised. The commonly used RTO calculation algorithms is, however, the basis for congestion control on most implementations today, and would be difficult to modify without risking a skew in throughput compared to unmodified TCP implementations. An alternative could be to implement a separate timer to handle thin-stream retransmission calculations (TCP Vegas-style), but that would require that we spend extra cycles on the separate timer calculations. We have therefore chosen to keep the RTO calculation unmodified.

Implementation of linear timeouts

When modifying the Linux code to implement linear timeouts for thin streams, we intercept execution at the point where the RTO is doubled as shown in figure 4.7. We insert an if-test that checks if the modification is enabled *tp->thin_rm_expb* || *sysctl_tcp_force_thin_rm_expb*, whether the stream is thin (*tcp_stream_is_thin(tp)*) and whether the connection is established (not in setup, teardown or a state of exception: *sk->sk_state == TCP_ESTABLISHED*). If the test is passed, we reset exponential backoff counters (*icsk->icsk_backoff = 0*) and perform a new RTO calculation (based on the last RTT measurement). This recalculation is performed because the stream might oscillate between being thin and not, thus keeping a very high RTO from before it qualified as thin. If any of the enabling requirements fail, the original code is

```
if ((tp->thin_rm_expb || sysctl_tcp_force_thin_rm_expb) &&
    tcp_stream_is_thin(tp) && sk->sk_state == TCP_ESTABLISHED) {
  /* Use linear timeouts */
  icsk->icsk_backoff = 0;
  icsk->icsk_rto = min(((tp->srtt >> 3) + tp->rttvar), TCP_RTO_MAX);
} else {
  /* Regular RTO calculation with exponential backoff */
  icsk->icsk_rto = min(icsk->icsk_rto << 1, TCP_RTO_MAX);
}
```

Figure 4.7: Code that enables linear timeouts if the stream is thin. Located in *net/ipv4/tcp_timer.c*.

called, which doubles the already calculated RTO (*icsk->icsk_rto « 1*) as long as the value does not exceed the maximum allowed RTO.

We need to confirm that the connection is in an established state since Linux TCP code uses the exponential backoff modifier to limit connection attempts in its three-way handshake. If a new RTO calculation is performed without any RTT measurements, a faulty (very low) RTO will be used (for the next connection attempt), and the connection will not be established.

4.1.4 Modified fast retransmit

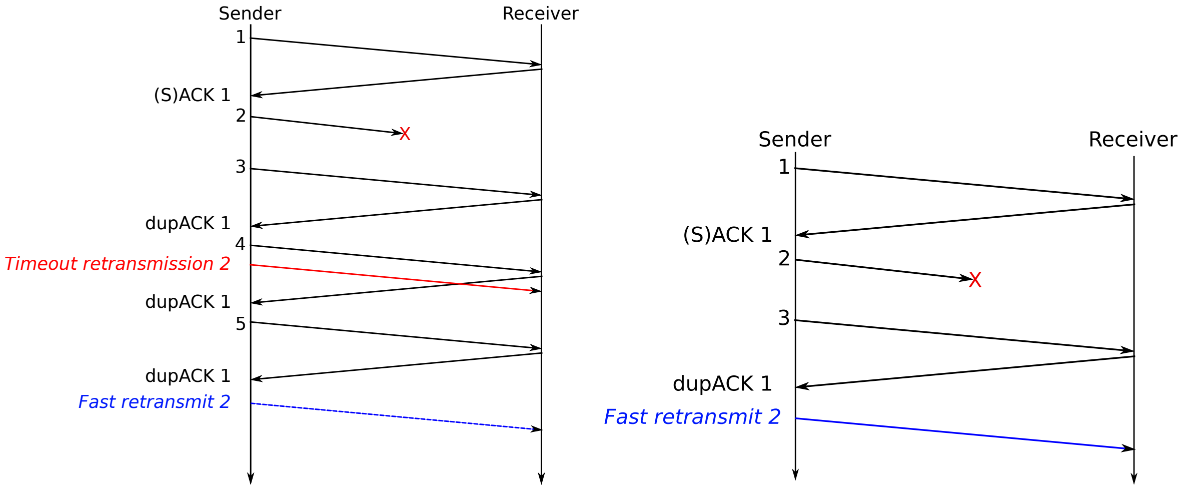
Due to the high interarrival time between data segments that the application produces, fast retransmissions hardly ever occur in our thin-stream scenario (as shown in section 3.9). In most cases, a retransmission by timeout is triggered before the three (S)ACKs that are required to trigger a fast retransmission are received. We therefore explore modifications to the fast retransmit mechanism in order to recover more quickly from loss when the stream is thin.

As for the other thin-stream mechanisms, the target of our investigations is to reduce the retransmission delay. Our analysis showed that a thin-stream sender often receives indications that loss or reordering has occurred (dupACKs), but rarely get enough to trigger the fast retransmit. We therefore wanted to experiment with modifications to this algorithm and analyse the effect they have on latency for the thin streams.

Choices

As the name indicates, fast retransmit was designed to recover from loss before a (slow) timeout was triggered. This was designed to happen on indications that packets were still leaving the network (incoming ACKs). In a scenario with many packets in transit, reordering are a relatively rare phenomenon[3]. With thin streams, however, this phenomenon will occur extremely seldom

[3] A study of reordering for web-server connections with 208.000 analysed connections showed that 0.04% of the packets were reordered [110]. For a few of the measured sites up to 3.2% reordering were observed. Such reorder rates were, however, categorised as anomalies.



(a) Fast retransmission after three dupACKs.

(b) Fast retransmission upon first indication of loss.

Figure 4.8: Difference between standard and modified fast retransmission for thin streams.

due to the low number of packet in transit. We therefore wanted to modify the number of dupACKs needed to trigger a fast retransmission. In order to reduce retransmission latency for a thin stream, we allow a fast retransmission to be triggered by the first indication that a packet is lost, as illustrated in figure 4.8. The thin stream, when recovering, does not expand its congestion window, and will therefore contribute little to renewed congestion. With regard to the retransmission latency, a fast retransmission that does not send the connection into slow-start and trigger exponential backoffs is preferable. In our opinion, the lowered retransmission latency justifies the need to risk occasional spurious retransmissions.

Implementation of modified fast retransmit

The function *tcp_time_to_recover()* contains a set of tests that determines whether a sign of congestion is caused by loss or reordering. If none of the tests returns *true*, the algorithm assumes that reordering has happened, and continues forward transmission. The code displayed in figure 4.9 is a new test that we have inserted at the end of the function. The test checks whether the thin-stream modification to fast retransmit is enabled (*(tp->thin_dupack* || *sysctl_tcp_force_thin_dupack)*, whether the stream is thin (*tcp_stream_is_thin(tp)*) and whether at least one dupACK has been received (*tcp_fackets_out(tp)* > *1)*. If all requirements are met, it signals initiation of recovery by returning “True” (1).

4.1.5 Redundant Data Bundling

As shown in table 2.1, many thin-stream applications send very small packets. For some of the analysed applications, the average packet size is below 100 bytes. In such a stream, a typ-

```
    if ((tp->thin_dupack || sysctl_tcp_force_thin_dupack) &&
            tcp_fackets_out(tp) > 1 && tcp_stream_is_thin(tp)){
            return 1;
    }
```

Figure 4.9: Code that adds a rule for thin-stream fast retransmission on first indication of loss. Located in *net/ipv4/tcp_input.c*.

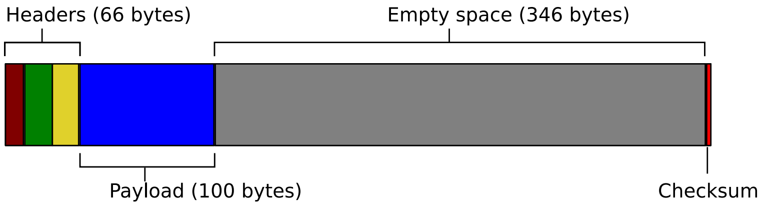


Figure 4.10: The minimum size of a Gigabit Ethernet frame when transporting a 100 byte packet [39].

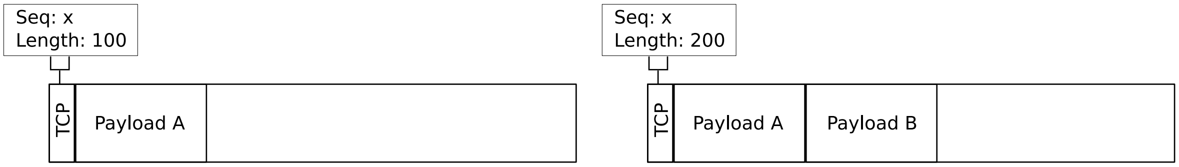
ical packet would consist of 40% headers (Ethernet header, Internet Protocol (IP) header and TCP header). Compared to the common maximum transmission unit of 1500 bytes (Ethernet 802.3 [58]), the utilised space would be no more than 11%. In high-speed networks, the minimum frame size is also increased so that maximum cable length would not be too short ([58]). As an example, Gigabit Ethernet uses a minimum frame size of 512 bytes (see figure 4.10). If each transmitted packet has a 100 byte payload, the utilisation would be 32%, leaving 68% of every sent packet empty (at least within the Gb Ethernet links[4]). The combination of these frame size considerations and the low rate of the thin streams led us to investigate possible ways to improve delivery latency by introducing an element of redundancy.

The aim of our investigations was to try to utilise the "free" space in thin-stream packets by resending unacknowledged data. This should be made in such a way that it would yield maximum improvement in delivery latency. The improved latency should be balanced against the potential redundancy the mechanism would create.

Choices

Our proposed mechanism is called redundant data bundling (RDB) and was refined and implemented in the Linux kernel [39]. As long as the combined packet size is less than the maximum segment size (MSS), we copy (bundle) data from unacknowledged packets in the send queue into the new packet. The copied data has to be in sequential order to comply with TCP standards, i.e. the oldest data must come first, and there may be no gaps in bundled packets. If a retransmission occurs, unacknowledged packets are bundled with the retransmission. This

[4] We think it fair to assume that the development of cabled networks will result in most network connections being high speed in time, considering that throughput still is the driving factor.



(a) First sent packet.

(b) Second packet: Bundled data.

Figure 4.11: Method of bundling unacknowledged data [87].

increases the probability that a lost segment is delivered with the following packet. Thus, a receiver may have the data delivered within a packet IAT, instead of having to wait for a retransmission. Figure 4.11 shows an example of how a previously transmitted data segment is bundled with the next packet. Notice how the sequence number stays the same while the packet length is increased. If packet (a) is lost, the ACK from packet (b) acknowledges both segments, making a retransmission unnecessary. In contrast to the retransmission mechanisms (modified dupACK and linear timeouts), which are triggered by there being fewer than four packets in transit, redundant data bundling (RDB) is enabled by small packet sizes, and limited by the packet IATs of the stream. If the packets are large, which is typical for bulk data transfer, bundles are not made and RDB not triggered.

An alternative to the chosen bundling strategy could be to bundle only on retransmissions (done to a certain degree by TCP already). We chose to let the mechanism bundle whenever possible, to maximise the effect on delivery latency. When the IAT is lower than the RTT, there are unacknowledged data to bundle for each new packet that is scheduled for transmission. This yields the lowest delivery latencies. When the IAT is too low, segments are combined until the MSS is filled. When the IAT is too high, ACKs are received before a new segment is transmitted. Alternative ways of adapting bundling to packet IAT or RTT are therefore not implemented. The mechanism does not bundle if the packet size exceeds the MSS. This automatically limits bundling to the small-packet scenario it was meant for.

RDB was developed as a modification to the Linux TCP stack. As such, it is transparent to the application. It is also designed to be transparent to the receiver. The only prerequisite is that a receiving TCP implementation must check *both* sequence number and packet size when a packet is received. As a TCP implementation needs to support this to be standards compliant, it is common practise for most operating systems today[5].

Implementation of RDB

The Linux implementation of RDB was first made in the 2.6.22.1 kernel, and incrementally improved and ported to the 2.6.23.8 kernel. In contrast to the modifications described in sections

[5] RDB has been successfully tested with Linux, FreeBSD, OS X (10.5) and Windows Vista computers as receivers.

4.1.3 and 4.1.4, which contained only small modifications to retransmission behaviour, RDB required more code. The code is included for inspection in appendix E. The main reason for the increased complexity was the need to copy between existing SKBs (Linux TCP segment control block). There is also need for extra accounting, since an incoming ACK may acknowledge half of the payload of a packet in the retransmission queue.

Figure 4.12 shows a flow diagram of the methods involved in the sending of packets using RDB. The methods that are new, or modified to facilitate RDB are marked in **bold**. Bundling can be attempted in two cases:

1. When the application produces data to be sent (for instance by calling *send()*).
2. When a retransmission is triggered by an incoming dupACK or a retransmission timeout.

When data is passed from the application, a new SKB is created. Bundling is enabled if the SKB passes the following test in *tcp_sendmsg()*: If the packet is smaller than the current MSS and contains no SYN or FIN flags, it calls *tcp_trans_merge_prev()*.

The check for SYN/FIN is made, because it is meaningless to bundle data in the setup and teardown process since no data is unacknowledged at that time. If the packet is found to be pure data, the previous SKB in the output-queue is checked to see if its payload can be copied into the current (most recent) SKB. If the new (bundled) packet is found to not exceed the MSS, the data from the previous SKB is inserted into the new one. In order to make room for the extra data, the current data is moved in memory. The new SKB inherits the sequence number from the previous SKB (reflecting the first byte in the payload), and its size is increased. When copying the data, care must be taken to handle linear and non-linear data in separate ways. This imposes a limitation on bundling, as all non-linear data must follow a linear data segment (of size *skb_headlen(skb)*) (see figure 4.13). Thus, if the current SKB has linear data, and the previous SKB has non-linear data, bundling is not performed. When an ACK arrives, the *tcp_data_snd_check()* function initiates a check for SKBs that are ready to be transmitted. If the ACK triggers a retransmission, and flow- and congestion control permits it, control is directed to the *tcp_retransmit_skb()* function. The TCP implementation enters this function also when a timeout is triggered. In *tcp_retransmit_skb()*, tests are performed to check that the SKB contains no SYN of FIN flags and if there are more outstanding (not acknowledged) SKBs. If these premises are fulfilled, *tcp_retrans_merge_redundant()* are called, and copy operations are performed similar to what was described for the function *tcp_trans_merge_prev()*. The RDB algorithm tries to include as much unacknowledged data as possible into the retransmitted packet. Therefore, the following SKBs are traversed until no more outstanding packets are found, or the packet size approaches the MTU. Since payload data is added *after* the current SKB in this case, the current payload will not have to be moved. This difference also means

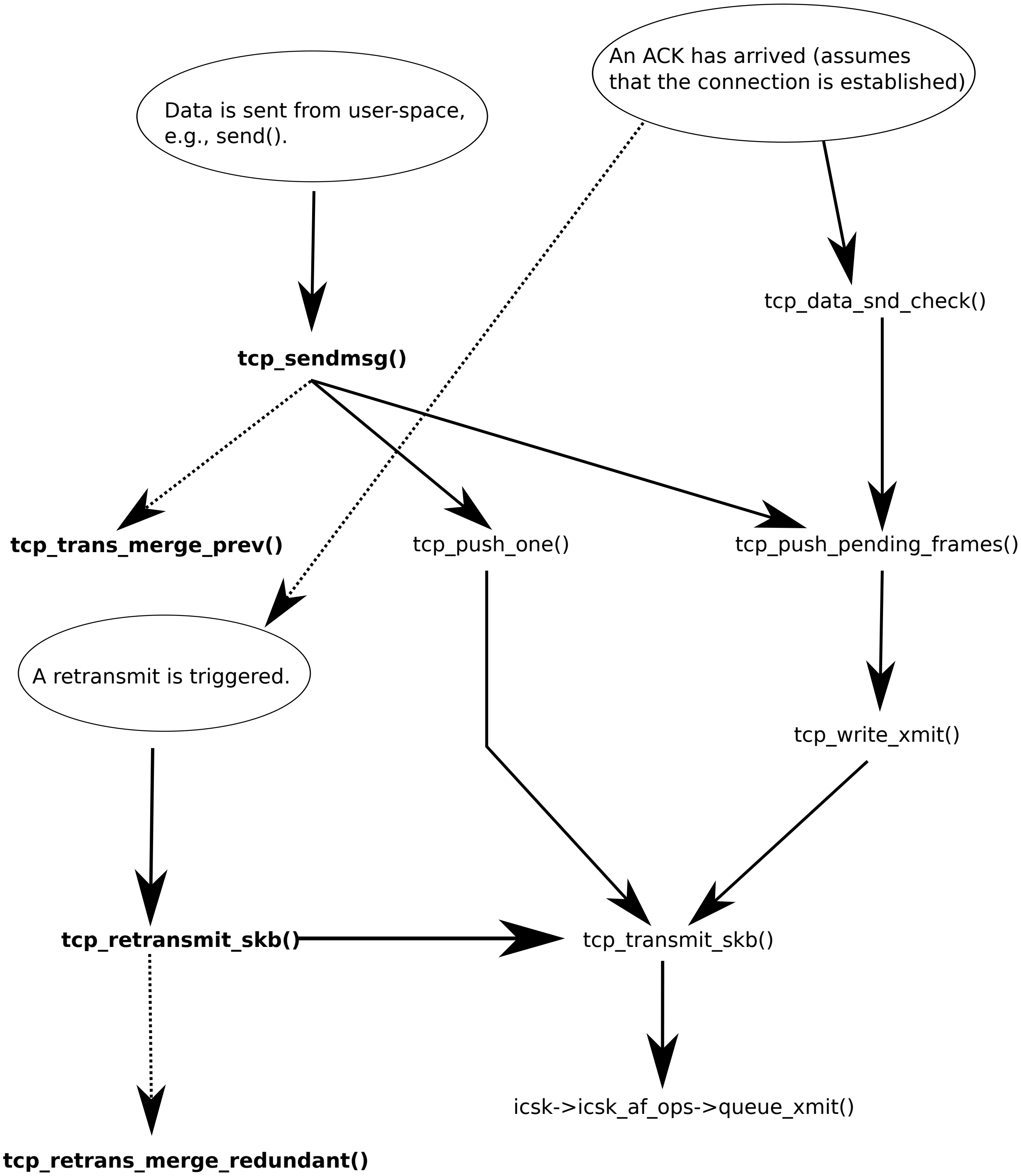
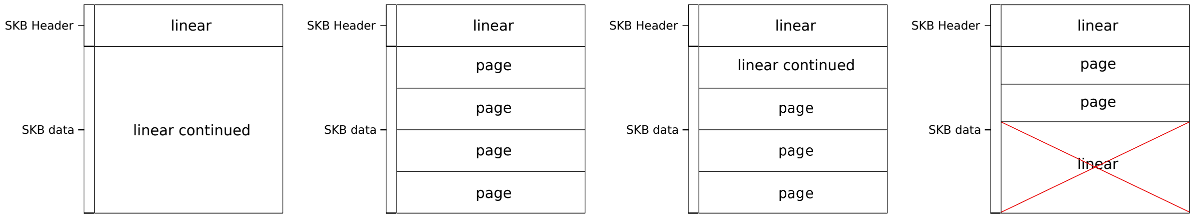

An ACK has arrived (assumes that the connection is established)
Data is sent from user-space, e.g., send().
tcp_data_snd_check()
tcp_sendmsg()
tcp_trans_merge_prev()
tcp_push_one()
tcp_push_pending_frames()
A retransmit is triggered.
tcp_write_xmit()
tcp_retransmit_skb()
tcp_transmit_skb()
icsk->icsk_af_ops->queue_xmit()
tcp_retrans_merge_redundant()

Figure 4.12: Flow-diagram: Outgoing TCP-packets. The functions we have implemented or modified are marked in bold [39].



(a) Linear memory SKB. (b) Non-linear memory SKB. (c) Linear/non-linear memory SKB. Allowed. (d) Linear/non-linear memory SKB. **Not allowed**.

Figure 4.13: SKB structure. Linear memory must all come at the start of the SKB.

that any linear data SKBs following a current non-linear SKB will prevent bundling (as shown in figure 4.13).

Since RDB modifies the internal structure of SKBs, accounting must also be done when ACKs arrive. Figure 4.14 shows a diagram of how the control flows when incoming packets arrive. The method that we have changed is **bold**. The control is passed from the IP layer to *tcp_clean_rtx_queue()* which removes acknowledged SKBs from the output queue. This method behaves exactly as unmodified TCP if the full payload of an SKB is acknowledged. The whole SKB is then removed from the queue. In order to improve accounting efficiency, avoid unnecessary data transmission and free space in SKBs for potential future redundant bundling, the handling of partially acknowledged packets is modified. We now remove acknowledged data from partially acknowledged packets.

4.2 SCTP modifications and implementation

SCTP does not only differ from TCP by being message-oriented, but also on subtle aspects of timers and retransmission techniques. The results from our initial evaluation of SCTP shows that it does not perform better than TCP New Reno with regard to latency for thin streams. As the SCTP retransmission mechanisms and implementation internals are different from TCP, the modifications also needed separate handling. In the next sections, we describe our SCTP-modifications for improving thin-stream latency, and their implementation in the Linux kernel.

4.2.1 Thin stream detection

SCTP differs from TCP by being chunk-oriented. This makes packet structuring more flexible than it is in TCP since the composition of chunks in a packet is not in any way predetermined. The mechanisms for triggering retransmissions are also based on per-chunk timing and statistics. To choose and implement a thin-stream detection mechanism in SCTP, different means had to be considered in order to reach the same goal of thin stream identification.

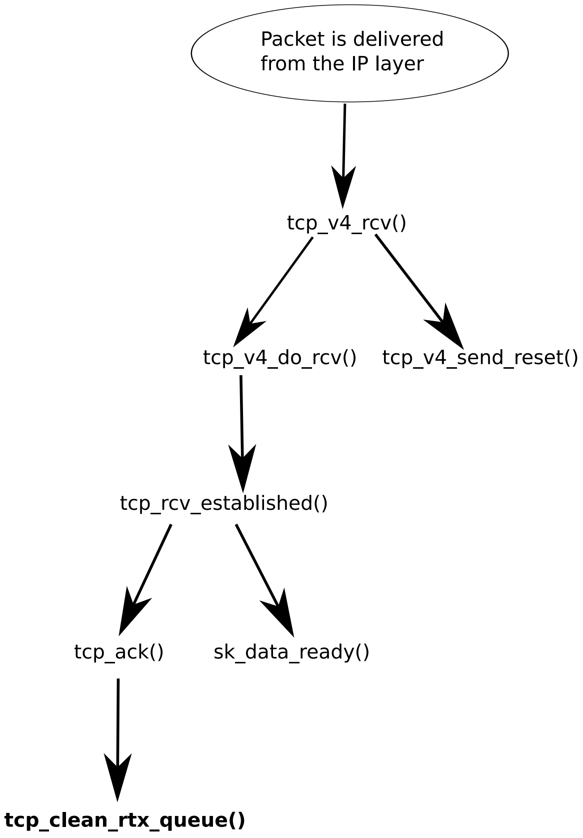


Figure 4.14: Flow-diagram: Incoming TCP-packets. The functions we have implemented or modified are marked in bold [39].

In the same manner as for TCP (section 4.1.1), we need to dynamically detect whether the stream qualifies as thin. There is need for effectiveness in the detection algorithm as the process is dynamic and continuous. Since SCTP can bundle several separate connections within an association, the metric used to identify if the stream is thin must be packet-based, not chunk-based. The method should be structured to be applicable for all connections within an association, thereby identifying connection-aggregated ability to trigger fast retransmissions. The mechanism should not have to rely on statistics passed from the application; it should be transparent to the application layer.

Choices

The number of packets in transit was the choice of metric in TCP, and we wanted to use a similar strategy for SCTP. After considering the available SCTP variables for accounting, we found no method that would provide equally good thin-stream separation as the TCP *in_flight* variable. In order to dynamically track thin streams, the accounting would therefore have to be separately implemented.

$$in\_transit \leq \frac{ptt_{fr} + 1}{1 - lossrate} \tag{4.2}$$

```
  struct pkt_stat{
    int is_gap_acked;
    int marked_lost;
    int indicated_lost;
    unsigned long timestamp;
    __u32 highestTSN;
    struct pkt_stat *next, *prev;
  };
```

Figure 4.15: Structure for keeping the information necessary to monitor loss rate and packets in transit. Located in *include/net/sctp/structs.h*.

The thin-stream mechanisms for SCTP was designed to follow the same scheme for triggering as shown in figure 4.1. The formula to determine if the stream is thin is shown in equation 4.2. Here, *in_transit* is the number of packets in flight (flight size), ptt_{fr} is the number of packets required to trigger a fast retransmission (3 according to RFC 4960 [95]) and *lossrate* is the fraction of packets that are detected as lost. The loss detection and modification was included as an experimental modifier to the packets in transit-test. The loss-modifier increases the thin-stream limit for *in_transit* if losses are high. This is to avoid the most severe cases of backoff that may happen in cases of extreme loss. One example of such loss is shown in [92], which presents measurements of VoIP traffic over large distances.

By using the *in_transit*-metric, we keep the limit for triggering the thin-stream mechanisms conservative, thus avoiding unwarranted aggressiveness in retransmissions. The mechanisms rely only on a count of unacknowledged, transmitted chunks, and do not take into consideration SACK or timing data.

Implementation of SCTP thin-stream detection

SCTP identifies chunks by transmission sequence number (TSN) and bundles them when retransmitting. The number of packets in flight is therefore not available, as it is in TCP. Thus, we added a list that holds the highest TSN for every packet in flight, as well as a packet counter. From the SACK, which acknowledges the highest cumulative TSN, the sender can now know whether or not a packet has left the network. Moreover, Linux kernel SCTP (lksctp) is not able to estimate packet loss, and therefore we implemented an algorithm for estimating packet loss that makes use of the packet-in-flight list to determine whether a packet is lost or not. Then, by looking at the SACKs returned by the receiver, we mark a packet as lost if the highest TSN in a packet corresponds to a gap in the SACK, and following the fast retransmission scheme, the packet is determined to be lost if it is indicated as lost by a SACK on three different occasions.

The structure displayed in figure 4.15 shows the variables needed in order to keep accounting for chunks that are lost or *in_transit*. *pkt_stat*-structs form a linked list, each representing a sent packet. The variable *is_gap_acked* indicates whether the chunk with the highest TSN in the packet was acked by a gap ACK (and therefore received). If *is_gap_acked* is set, the struct will still be kept in the list until the chunk is cumulatively acked. *marked_lost* is set if the packet can be classified as lost using TCP metrics (three duplicate SACKs or a timeout). The *indicated_lost* variable keeps count of dupACKs that signify loss indication. A *timestamp* is kept that represent the time the packet was sent. Finally, the *highestTSN* keeps the highest TSN of the chunks in the current packet. The highest TSN is used to represent a sent packet, since SCTP uses bundling strategies that do not tie specific chunks to certain packets.

The list of *pkt_stat*-structs is accessed from the *sctp_association*-struct in *include/net/sctp/structs.h* and updated on construction of each SCTP packet. The *sctp_association*-struct also has variables for counting the packets in transit and the number of packets that has left the network. When a SACK arrives that cumulatively ACKs the *highestTSN* of one or more elements of the *in_transit*-list, the element is removed, and the number of *in_transit* packets decremented.

Identification of reason for retransmission

SCTPs various methods of chunk bundling enable uncountable variations of chunk stacking upon retransmission. To determine the reason for each retransmission by trace analysis alone would therefore be a very complicated task. We, however, do not need to analyse the data from the transmissions. Our field of interest is the packet IAT and size. We therefore implemented an optional mechanism that “tags” the first byte of all chunks in every retransmitted packet with the reason for retransmission. The tagging is a special feature that allows us deeper insight into which retransmission mechanisms that are dominant for each tested scenario. The chunk tagging does, however, render the payload useless for an application and is therefore only of value for analytical purposes.

Figure 4.16 shows the code that performs this tagging. First (line 1-2), the payload of the chunk is located, the offset indicating the first byte of chunk data. The byte is then replaced with a character indicating the reason for this chunk’s retransmission: ’f’ means fast retransmit (line 4), ’t’ means retransmission by timeout (line 8) and ’b’ is a bundled chunk (line 10). To determine which chunk triggered a retransmission by timeout, the lowest TSN of all chunks in the packet is found (line 7). The chunk with the lowest TSN is the one that triggered the timeout, the rest of the chunks in the packet are bundles.

4.2.2 Implementation of switches to enable thin-stream modifications

In our modifications to SCTP, we implemented enabling of each mechanism through dedicated syscontrols. Extended per-application flexibility through *IOcontrols* are not implemented for our SCTP modifications due to time limitations.

```
#ifdef THIN_ANALYSIS
data_chunk_payload = (char *) chunk->skb->data;
chunk_offset = 16; // first byte of data chunk payload
if(fast_retransmit){
  data_chunk_payload[chunk_offset] = 'f';
} else {
  currentTSN = ntohl(chunk->subh.data_hdr->tsn);
  if(currentTSN == lowestTSN){
    data_chunk_payload[chunk_offset] = 't';
  } else {
    data_chunk_payload[chunk_offset] = 'b';
  }
}
#endif
```

Figure 4.16: Code for tagging the payload of packets with the reason for retransmission. Only used for the purpose of analysis as it will overwrite the chunk payload. Located in *include/net/sctp/outqueue.c*.

Mechanism	**sysctl**	**Input**
modified RTO$_{min}$	sctp_thin_minrto	Bool
modified timer restart	sctp_thin_restart_timer	Bool
LT	sctp_thin_expbackoff	Bool
mFR	sctp_thin_fr	Bool
bundling on fast retransmit	sctp_thin_bundling_fr	Bool
chunk tagging	sctp_thin_debug_tag_payload	Bool

Table 4.2: Mechanisms implemented for triggering the modifications in SCTP.

Table 4.2 shows a list of the syscontrols that can be used to enable our implemented mechanisms. The “Input”-column shows the parameters given to the control. “Bool” signifies an on/off trigger. No numerical limits were implemented for our SCTP modifications, and so all the switches are boolean. Examples of how the modifications are enabled can be seen in figure 4.17. The *sysctl*-command can be used as shown in figure 4.17(a). An alternative way is to write the value to the *proc*-file system variable for this syscontrol as shown in figure 4.17(b). Root access is be needed to set such system-wide parameters.

4.2.3 Modified minimum retransmission timeout

To avoid timeouts occurring too early, which can lead to spurious retransmissions and a reduced congestion window, SCTP has a rather high RTO$_{min}$ value (1000 ms). Nevertheless, in our thin-stream scenario, we can see that almost all retransmissions are due to timeouts. We therefore wanted to investigate the effect of changing the SCTP RTO$_{min}$.

When the stream is detected to be thin, the high RTO used in SCTP brings no value to the

```
sysctl net.ipv4.sctp_thin_minrto=1
```

(a) syscontrol enabling of modification

```
echo “1” > /proc/sys/net/ipv4/sctp_thin_minrto
```

(b) /proc-variable enabling of modification

Figure 4.17: Examples of how to enable the modified RTO$_{min}$ thin-stream modification.

connection. We therefore want to reduce the RTO$_{min}$ value to lower the retransmission delay when timeouts happen.

Choices

Since our goal is to make small, effective modifications to improve latency for thin streams, we find it more expedient to make an adjustment to the RTO$_{min}$ than to try to develop a whole new RTO calculation algorithm. The RTO$_{min}$ is a fixed value, which represents the absolute minimum the system should wait before retransmitting if no feedback is received. An RTO$_{min}$ that is too low could result in a drastic increase in spurious retransmissions. An RTO$_{min}$ that is too high could (as shown in section 3.4) cause very high retransmission latencies for thin streams. Our choice fell on the well-proven 200 ms RTO$_{min}$ used in the Linux kernel implementation of TCP. This value allows for feedback to arrive for most connections, while not delaying retransmissions unacceptably long (in an interactive application's perspective).

As a consequence of reducing RTO$_{min}$, the relative effect of delayed SACKs on the RTO calculation that was described in section 3.4 grows. When the receiver-side SACK delay is eliminated, the calculated RTO is greatly reduced due to a lower measured RTT. Thus, although receiver-side enhancements are more difficult to apply in some scenarios (since client machines must be updated), we experimented also with the disabling of delayed SACKs.

Implementation

The code displayed in figure 4.18 controls the value that is used as RTO$_{min}$ for the given SCTP transport stream. If the *syscontrol sctp_thin_minrto* is set, and the stream is identified as thin using the mechanisms described in section 4.2.1, the modified code is enabled. If the calculated RTO (*tp->rto*) is lower than 200 ms, the RTO$_{min}$ is set to 200 ms. If the stream is not thin, or the *syscontrol* is not enabled, the system RTO$_{min}$ of 1000 ms is used. If a greedy stream becomes thin, a new RTO calculation needs to be performed before the modification takes effect.

```
if(sctp_thin_minrto &&
    (tp->asoc->packets_in_flight <
     tp->asoc->thin_stream_threshold) ) {
  if(tp->rto < msecs_to_jiffies(200)){
     tp->rto = msecs_to_jiffies(200);
  }
} else {
  if (tp->rto < tp->asoc->rto_min){
      tp->rto = tp->asoc->rto_min;
  }
}
```

Figure 4.18: Code to reset RTO$_{min}$ to 200 ms if the stream is identified as thin. Located in *include/net/sctp/transport.c*.

4.2.4 Correcting the RTO timer reset

When SCTP receives a SACK that acknowledges some, but not all, of the outstanding chunks, the RTO timer is reset. For greedy streams, this increases the probability for a retransmission to be triggered by a fast retransmit before a timeout happens, thus saving the stream from a slow start that reduces the overall throughput. For thin streams, however, the chance of triggering a fast retransmit is so small that this mechanism creates additional delay with no prospect of gain. We have therefore investigated ways of modifying SCTP in order to improve thin-stream latency for this special case.

We want to reduce the probability that increased latency occurs when SACKS arrive that acknowledges only some of the outstanding chunks. This could happen in any of our described thin-stream scenarios, and should therefore be used in conjunction with any of the other thin-stream mechanisms that we describe here. The timer reset can cause extra retransmission delay for interactive applications whether or not the stream is thin. We therefore want to be able to enable the modification for all SCTP associations if the system administrator knows that the machine will serve interactive applications.

Choices

Figure 4.19 shows an example of how the timer is reset if a SACK arrives that acknowledges only some of the outstanding chunks. The chunk with TSN 100 is sent at time 0 over a 200 ms RTT association. After 250 ms, the chunk with TSN 101 is sent and lost. Because delayed SACKs are standard in SCTP, the TSN 100 is not acknowledged until after 400 ms. At the time when the SACK for TSN 100 is received, the TSN 101 chunk is still outstanding. Despite of this, the RTO is reset to the RTO$_{min}$ of SCTP of 1000 ms. The effect of this timer reset is that the chunk with TSN 101 that would have been retransmitted at time 1250 ms without the reset is actually retransmitted at 1400 ms. The added delay for this example is 150 ms (an increase

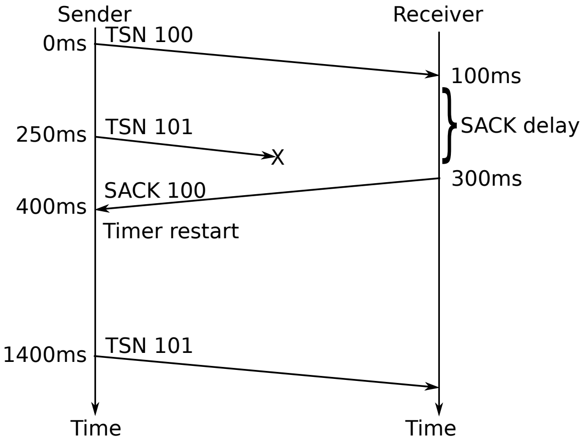


Figure 4.19: Example of how resetting the RTO can increase retransmission delay for thin streams [83].

of 13%).

In order to improve this situation, we have to choose a way of correcting the RTO restart when a SACK is received that acknowledges only some of the outstanding chunks. When we consider that we also propose other modifications that involve the RTO timer, we want to do this adjustment relative to the RTO value that is calculated for the current stream and configuration. We therefore explored ways of subtracting the "penalty" that exceeds the RTO that would have been used for the chunk, had not the timer reset been performed. A suggested way of handling this situation is presented in [67]. According to the paper, a correct RTO can be ensured by subtracting the age of the earliest outstanding data chunk from the current RTO value each time the timer is restarted. This approach is used as the basis for our modifications to the timer recalculation.

Implementation

The function *sctp_check_transmitted()* is located in *outqueue.c*, and performs a traversal of outstanding chunks when a SACK arrives. It also performs adjustments to the RTO timer if warranted. The code that performs the correction to the RTO timer restart is called if the variable *restart_timer* is set, signalling the RTO recalculation. Figure 4.20 shows the modified code to be called when this recalculation is triggered. The premise for using the modified code is that the syscontrol *sctp_thin_restart_timer* is set in line 1. If this is not the case, the original timer reset is performed (lines 22-25). The first step is to find the age of the oldest outstanding chunk. This is performed in lines 3-12 by traversing the transmitted-queue and selecting the first chunk in the queue that is not gap ACKed. If such a chunk is found, the time since the chunk was sent is subtracted from the current RTO (lines 13-19). The method *mod_timer()* performs the recalculation of the RTO. If the timer is successfully set, the execution is continued (by exiting the function).

```
if(sctp_thin_restart_timer){
  oldest_outstanding_chunk = NULL;
  list_for_each(list_chunk, &tlist){
    cur_chunk = list_entry(list_chunk, struct sctp_chunk,
    transmitted_list);
    if(sctp_chunk_is_data(cur_chunk)){
      if(!cur_chunk->tsn_gap_acked){
        oldest_outstanding_chunk = cur_chunk;
        break;
      }
    }
  }
  if(oldest_outstanding_chunk != NULL){
    if (!mod_timer(&transport->T3_rtx_timer,
    (jiffies -
    (jiffies - oldest_outstanding_chunk->sent_at)
    + transport->rto))){
      sctp_transport_hold(transport);
    }
  } else {
    if (!mod_timer(&transport->T3_rtx_timer,
    jiffies + transport->rto)){
      sctp_transport_hold(transport);
    }
  }
}
```

Figure 4.20: Code to avoid restarting the retransmission timer when a SACK arrives. Located in *include/net/sctp/outqueue.c*.

4.2.5 Linear retransmission timeouts

If there are too few SACKs to trigger a fast retransmission or no new packets are sent to let the receiver discover loss, retransmissions are triggered by subsequent timeouts without any intervening fast retransmissions. At this point, an exponential back-off of the retransmission timer is performed as for TCP, which leads to the retransmission delay increasing exponentially when there are occurrences of multiple loss. This effect is aggravated by the high RTO_{min} used by SCTP. We therefore investigated modifications to the SCTP exponential backoff mechanism for thin streams.

Given that the stream is identified as thin, we want to modify the existing exponential back-off mechanism so to reduce the extreme retransmission latency that can arise due to this mechanism. Alternative ways of calculating the RTO could have been devised, but as for our TCP modification described in section 4.1.3, we wanted to keep the changes as simple as possible. If the stream is not thin, the original SCTP-mechanisms are effective due to more frequent feedback, and should therefore be kept.

```
1 if( !(sctp_thin_expbackoff &&
2       asoc->packets_in_flight >=
3       asoc->thin_stream_threshold) ) {
4   transport->rto = min((transport->rto * 2), transport->asoc->rto_max);
5 }
```

Figure 4.21: Code to use linear timeouts if the stream is identified as thin. Located in *include/net/sctp/sm_sideeffect.c*.

Choices

The actual effect of exponential backoff is reflected by the value of the RTO$_{min}$. In the case of SCTP, the 1000 ms RTO$_{min}$ yields a high starting point for exponential backoff for any connection. When the problems with RTT estimation caused by delayed SACKs (described in section 3.4) are taken into consideration, retransmission latencies are prone to be very high. We therefore suggest that linear timeouts are applied when the stream is detected as thin. The effect is then the same as for our linear timeout-modification for TCP (illustrated in figure 4.6). The linear-timeouts modification is recommended to be used in conjunction with a reduced RTO$_{min}$ (described in section 4.2.3). We considered alternative schemes for calculating the RTO, but concluded that a simple modification only in the cases where the stream is thin was preferable. The exponential backoff is not needed to protect against aggressive probing for bandwidth, since the stream is thin, and has only small amounts of data to transmit.

Implementation

lksctp handles exponential backoff of the retransmission timer in the function *sctp_do_8_2_transport_strike* which is located in *sm_sideeffect.c*. Our modification consists of a wrapper around the statement that performs the doubling of the RTO. Figure 4.21 shows the code that performs the modification. Line 4 performs a doubling of the RTO, or sets it to RTO_{max}. In line 1, 2 and 3, a check is performed to determine if the modification is enabled (by *sctp_thin_expbackoff*), and whether the stream is thin using the mechanisms described in section 4.2.1. If the tests confirms the stream as thin and the modification is enabled, no change to the RTO is performed. If any of the test criteria fail, the RTO is doubled as normal.

4.2.6 Modified fast retransmit

Despite the high RTO$_{min}$ value, fast retransmissions hardly ever appear in our thin-stream scenario. If only two or three packets are sent every second, even a one second RTO$_{min}$ is often triggered before a fast retransmission. The delayed SACKs that obfuscate RTO calculation for thin streams (as shown in section 4.2.3) reduce the chances of triggering fast retransmissions further. We have investigated this mechanism in SCTP and implemented modifications to the fast retransmit mechanism to reduce the retransmission delay for thin streams.

In the same way as for TCP (described in section 4.1.4), we want to find ways to relax the fast retransmission dupACK requirement to help reduce the retransmission delays. This mechanism should also be dynamically triggered *only* if the stream is identified as thin.

Choices

When investigating changes to the fast retransmit mechanism, we first had to consider the basic rationale behind the mechanism. The mechanism was designed to use the indications of loss in the form of duplicate ACKs to recover without having to go into slow start, which again would lead to exponential backoffs if the chunk was lost again). We have shown in section 3.4 that thin streams have very low probability to trigger a fast retransmit even when the RTO$_{min}$ is as high as 1000 ms. We wanted to keep the properties of fast retransmit (of avoiding slow start). The fast retransmit mechanism's effect on the congestion window does not have any effect for thin streams since the do not expand it. We therefore considered the most effective solution for fulfilling our goal without disturbing regular SCTP operation to allow the threshold for a fast retransmit to be lowered from three dupACKs to one (as illustrated in figure 4.8. The modification implemented may lead to more transmissions in the low-probability case that packets are reordered, but the gain in latency justifies the need to drop occasional spurious retransmissions.

Implementation

The file *outqueue.c* holds the method that traverses the transmitted-queue, and marks data chunks as missing based on SACK information (*sctp_mark_missing*). Figure 4.22 shows the code that implements the modification. Lines 9-12 hold the code that triggers a fast retransmit when the limit of three dupSACKs is reached. We have removed the hard limit of three dupACKS and included a test that dynamically sets the limit to one if the stream is identified as thin and the mechanism is enabled (lines 1-7). To determine whether the stream is thin, we access the implemented mechanism through the *asoc*-struct which is referenced by the transmitted-queue (*q*).

4.2.7 Bundling on fast retransmit

SCTP bundles outstanding chunks with a retransmission by timeout if there is room in the packet (limited by the system MTU). For a greedy stream, a retransmission by timeout is an indication of heavy loss, since too few dupACKs have been returned to trigger a fast retransmit and signal that packets have left the network. For thin streams, a timeout may well happen even if only one

```
if(sctp_thin_fr){
  if(q->asoc->packets_in_flight < q->asoc->thin_stream_threshold){
    fr_threshold = 1;
  } else {
    fr_threshold = 3;
  }
}

if (chunk->tsn_missing_report >= fr_threshold) {
  chunk->fast_retransmit = 1;
  do_fast_retransmit = 1;
}
```

Figure 4.22: Code to modify the fast retransmit threshold if the stream is identified as thin. Located in *include/net/sctp/outqueue.c*.

packet is lost, since it often cannot trigger a fast retransmit in any case due to the high packet IAT. When a fast retransmit is performed, SCTP does not bundle any outstanding chunks except for chunks that are already tagged for fast retransmit. We have therefore investigated bundling of unacknowledged chunks upon fast retransmit in order to improve delivery latency for thin streams. By allowing the sender to bundle unacknowledged chunks upon fast retransmit, we aim to improve the delivery latency when loss occurs.

Choices

We have chosen to implement a mechanism that tries to bundle as many chunks as possible upon a fast retransmit until the MSS is reached. This is done only when the stream is detected as thin. If this option is used without also enabling the modification that allows a fast retransmit on the first dupACK, it will have small (or no) effect. Another option could be to enable bundling on fast retransmit without limiting the effect to thin streams. This would increase the amount of spurious retransmissions, but could potentially improve the delivery latency for borderline cases of thin streams (low IAT, but not low enough to trigger pass the conservative test described in section 4.2.1). An effect of such bundling would, however, be that the congestion window would continue to grow even if loss had occurred. Greedy streams would seldom cause spurious retransmissions if bundling on fast retransmissions were enabled, since the MSS would nearly always be reached.

Implementation

We have implemented a new function to replace the test that was originally performed in *sctp_retransmit_mark()* to determine whether a fast retransmit should be performed. The new function is *check_stream_before_add()*, and returns true if bundling should be performed. The code

```
1  int check_stream_before_add(struct sctp_transport *t,
2                              struct sctp_chunk *chunk,
3                              __u8 fast_retransmit){
4    if(sctp_thin_bundling_fr){
5      if(t->asoc->packets_in_flight < t->asoc->thin_stream_threshold){
6        return ((fast_retransmit && chunk->fast_retransmit) ||
7                !chunk->tsn_gap_acked);
8      }
9      else{
10       return ((fast_retransmit && chunk->fast_retransmit) ||
11               (!fast_retransmit && !chunk->tsn_gap_acked));
12     }
13   }
14   else{
15     return ((fast_retransmit && chunk->fast_retransmit) ||
16             (!fast_retransmit && !chunk->tsn_gap_acked));
17   }
18 }
```

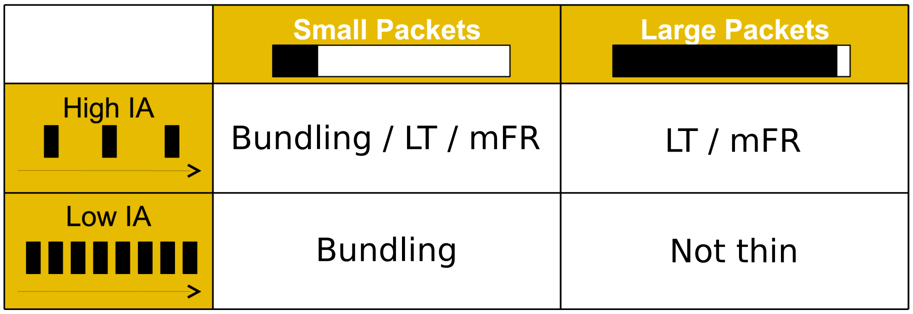
Figure 4.23: Code to allow bundling of unacknowledged chunks on fast retransmit if the stream is identified as thin. Located in *include/net/sctp/outqueue.c*.

that performs the bundling itself is not included here as it conforms with the original SCTP bundling implementation. Figure 4.23 shows the modified code. The *sctp_transport*-struct is passed to the function for access to the *asoc*-struct where the thin stream limit information is kept. References to the current chunk and fast_retransmit status is also passed. In line 4, the code checks whether bundling on fast retransmit is enabled by the *syscontrol*. The code in line 6 checks whether the stream is currently thin. If the stream is thin, the chunk is tagged for bundling if it is not already confirmed delivered by a gap ACK (lines 6 and 7). If the modification is not enabled by the *syscontrol* or the stream is not identified as thin, only chunks tagged for fast retransmit are bundled.

4.2.8 Other possible avenues of investigation

SCTP has a very flexible framework for allowing bundling of chunks. Since each chunk has a unique TSN, the chunks from different applications can be interleaved and chunks that are not in consecutive order can be retransmitted in the same packet. This freedom makes the possibilities for inventive bundling mechanisms near limitless.

Within the time limit and scope of this thesis, the previously described modifications have been implemented and tested. Other mechanisms, like preempting loss by bundling all outstanding data on regular transmissions (much like RDB in TCP described in section 4.1.5) have also been discussed but not fully implemented and tested due to time limitations. Other schemes like transmitting each chunk twice within a certain time limit (if packet space allows) could also



	Small Packets	Large Packets
High IA	Bundling / LT / mFR	LT / mFR
Low IA	Bundling	Not thin

Figure 4.24: Applicability of the thin-stream mechanisms for TCP.

be explored in future work.

4.3 Applicability of modifications

All of our developed mechanisms, both for TCP and SCTP, are basically targeted at improving latency for thin streams. As thin streams can vary in packet IAT and size, the different mechanisms perform differently based on the current stream properties.

In figure 4.24, we characterise the main modifications based on the thin-stream properties that they are applicable for. Some of our modifications, like the reduced RTO$_{min}$ and corrected timer reset for SCTP are applicable for all delay-sensitive, thin-stream applications, regardless of the degree of packet size and packet IAT within the thin-stream criteria, and are therefore not included in the figure.

The typical thin-stream case is displayed in the upper left corner of figure 4.24; small packets and high packet IAT. Such properties can be seen in the analysis of the MMORPGs (WoW, AO and AoC), RTS game (WiC), SSH-session, sensor network and RDP from table 2.1. Bundling is possible for this class of streams since the low packet size makes room for bundles before reaching the MSS. The high packet IATs reduces the chance that fast retransmissions are triggered. Applying the linear timeouts (LT) and modified fast retransmit (mFR) therefore helps to reduce the retransmission delay.

The class of streams described in the lower left quadrant of figure 4.24 has small packets, but relatively low packet IATs. Where a greedy stream would always fill the MSS of each packet, the packet IAT for this class of stream is still so high as to allow the small packet sizes. For this class of applications, bundling schemes greatly helps to reduce the delivery latency when loss occurs. Aggressive bundling schemes like RDB can, for borderline thin-stream cases, quickly fill up the packet MSS, thus consuming more bandwidth. The number of packets sent will, in most cases, not be drastically increased. There may be a increase in the number of sent packets due to the fact that aggressive bundling may cause a segment to be ACKed that would otherwise be lost. This helps the connection to avoid a timeout with following slow-start(s) and eventual exponential backoffs. In practise, this also means that the number of sent packets is sometimes somewhat higher.

The upper right quadrant of figure 4.24 represents a special case that occurs, but that is not common. A high packet IAT combined with large packets can occur if the application has a transmission timer for certain events, and that the data to be sent has a size that is close to the network MSS (or above the MSS, resulting in multiple packets). An example could be a video codec producing frames at certain intervals with a given packet size. In table 2.1, the closest example is from the RTS “World in Conflict”. The *from server*-stream here has a packet size of 365 bytes on average, and transmits 10 packets each second. For this class of applications, the LT and mFR mechanisms are beneficial, but bundling rarely happens.

The last quadrant in the figure (bottom right) represents a typical greedy (or non-thin) stream. In this case, standard congestion-control and retransmission mechanisms are functional with regard to latency, and should be used in their unmodified version.

Chapter 5

Analysis and evaluation

Performing a well-balanced evaluation of the modifications described in chapter 4 required that we chart several key properties of the behaviour of traffic transmitted using the modifications:

- The effect of each modification on latency for different network conditions.
- The effect of combinations of modifications.
- The effect of the modifications on unmodified, competing streams.

In order to answer these questions, we had to be able to control the network parameters (RTT, loss rate and loss burstiness) and also to control the behaviour of the stream (packet IAT, packet size). Most network emulators (like netem [1]) causes evenly distributed loss. In the Internet, loss patterns caused by congestion is more bursty. In order to answer the questions we had as accurately as possible, we set up a several different test environments, and performed exhaustive tests. This chapter describes the test environments, the tests that were performed and the results from the different tests.

5.1 Test environment, evaluation setup, tools and metrics

We needed to control the network properties to gather information about the behaviour of the modifications under different circumstances. For this purpose, laboratory environments were the best way to perform controlled tests for each combination of network and stream properties. To gather data about the performance of the modifications under real-life Internet conditions, we also set up several tests that transmitted data over the Internet. All experiments that are directly compared have been performed on the same hardware with the same system configuration[1].

[1] We have tried to achieve configuration equality as far as can be done. Different operating systems do, however, have slightly different options. In such cases, we have striven to match system settings as close as possible within the respective system's limitations.

The sender used our modified Linux kernel when testing the modifications, the receiver always used an unmodified kernel. The reason why the receiver always used an unmodified kernel is the requirement that our modifications should be transparent to the receiver. By adhering to the TCP standards, any host with a correct TCP implementation should be able to receive a stream with or without modifications. The following sections describe each of the main test environments used to evaluate the TCP, SCTP and UDP/application layer mechanisms.

5.1.1 Alternative evaluation methods

Our proposed mechanisms were implemented in the Linux kernel. The effects were then evaluated by performing detailed laboratory tests and Internet tests. The possibility of performing an evaluation by simulation was explored, but rejected. It proved to be difficult for the following reasons:

- We could not find a good framework that could be reliably modified to simulate the thin-stream behaviour for our modifications. The main problem was that simulators (like ns2) are usually focused on throughput and congestion control for greedy streams. To reduce the complexity of simulations, they use an abstraction of packet size, assuming that all segments are filled to the MSS. A reliable thin-stream simulation would therefore be impossible without making major changes to the simulator structure.

- Simulation frameworks that use the Linux kernel as a base for their networking were evaluated. The tested frameworks did, however, either abstract segment sizes like explained in the previous item, or proved difficult to integrate with our thin-stream modifications due to special timer adaptations made for the simulator. In the one case [98] where simulations looked promising, an error in the simulator's network modelling invalidated the gathered results.

The alternative of developing our own simulation framework would be time-consuming and still lack the confidence provided by a well-tested framework. We therefore concluded that the (potential) benefits of speed-up of test execution, large-scale network simulations and fairness evaluation in a simulated environment would not outweigh the disadvantages.

5.1.2 Laboratory tests with artificial loss

The experiments that aim to chart the behaviour of our thin-stream modifications under different network conditions demand that we have total control over both network and stream properties. In the network, we want to be able to control and vary RTT and loss rate. In order to test for different aspects of thin streams, we must be able to control packet IAT and size. We must also be able to record the resulting data patterns, both on the sender and on the receiver.

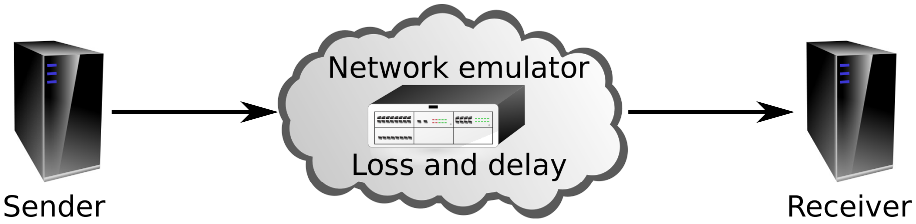


Figure 5.1: Test setup where a network emulator is used to create loss and delay.

An identifying property of thin streams is the high packet IAT. In order to generate enough instances of loss to get statistically viable numbers, tests have to be run for a long time[2]. Since jitter has shown to influence the latency very little when TCP New Reno is used (see section 5.2), we have chosen not to include tests with jitter variations.

The layout of our laboratory test environment for artificial (network emulator) loss is shown in figure 5.1. The evaluated streams are simplex (one-way) flows from the sender to the receiver. The data is transmitted to the receiver via a network emulator which induces loss and delays the packets. Loss and delay are induced in both directions for the connection, meaning that ACKs are also lost (as may happen in live scenarios). Loss rates and delay are presented for each performed test. Packet traces are recorded at the sender, and also at the receiver if required for test analysis. This test setup generates a predictable loss rate with packet losses occurring at regular intervals.

5.1.3 Laboratory tests with cross-traffic induced loss

In order to get an indication of how thin streams perform in a more realistic loss scenario, we created another laboratory test environment. The aim was to create loss in the same manner as on a congested Internet link; through a bandwidth-limited node with a tail-dropping queue and traffic that causes congestion.

Figure 5.2 shows the layout of our cross-traffic test setup. The sender and receiver computers behave like described in section 5.1.2. Delay, both upstream and downstream, is still created by the network emulator. It does, however, no longer drop packets, but instead uses traffic control[3] to enforce bandwidth limitation with a tail-dropping queue on the pass-through link. We use two extra machines to generate cross traffic across the bandwidth-limited link.

The cross traffic generation was designed to mimic HTTP-like traffic. With respect to emulating such traffic, a lot of work has been done to define parameters such as file-transfer size and mean interarrival time, as well as the number of concurrent clients [29, 19]. Most studies

[2] If a thin stream has a steady packet IAT of 120 ms, and the link loss rate is 2%, a test has to be run for 8 hours in order to generate 4800 instances of loss.

[3] Linux *tc* with *netem* was used [1]. The link was limited to 10Mbps with a "hierarchical token bucket"-algorithm using the following *tc*-commands: "tc qdisc add dev eth5 root handle 10: htb default 1", "tc class add dev eth5 parent 10: classid 10:1 htb rate 10000kbit ceil 10000kbit burst 6k prio 2"

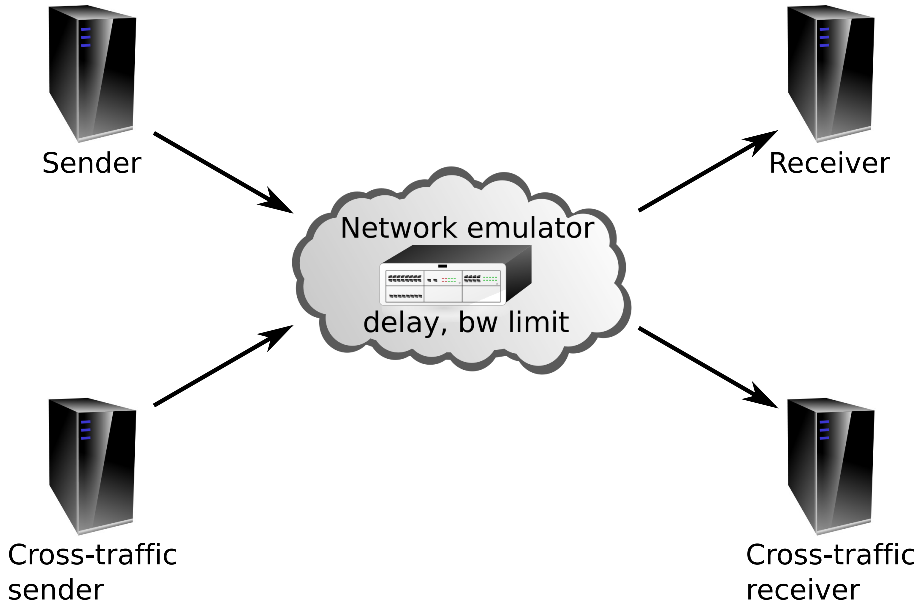


Figure 5.2: Test setup where a network emulator is used to create delay and limit bandwidth. Loss is created by competing HTTP-like cross-traffic.

agree on a heavy-tail distribution to describe the file sizes [29]. The studies show that there are many small files, and few large ones, but the greater sizes can become almost arbitrarily large. Thus, we used a Pareto distribution with a minimum size of 1000 bytes[4] giving us a mean transfer size of approximately 9200 bytes per connection. Furthermore, we had 81 concurrent web-client programs running, where the number was determined by the number of different delays that one *netem* instance can assign to connections. Each of the client programs started new streams within a pre-configured time-interval. To control the loss rate, we set the request interarrival-times to follow an exponential distribution with a preset mean value. On the bottleneck, the bandwidth was limited to 10Mbps with a queue length of 100 packets. The queue length was decided after experimenting for parameters that would give a realistic loss scenario. A longer queue would lead to bursts of delivered packets with large jitter, while a shorter queue resulted in high loss rates.

5.1.4 Internet tests

The performance of our modifications in a live Internet scenario was tested using the basic setup shown in figure 5.3. In order to get an impression of the network characteristics for different types of applications (for instance p2p and client/server), we varied which ISP the sender and receiver were placed at. In such tests, loss and delay varies depending on path properties and

[4] The maximum size was limited to approximately 64 MB in our cross-traffic environment. If we were to allow arbitrarily large file sizes, given the configured bandwidth limitation, the large files would, over time, dominate the traffic, and the desired effect would be lost.

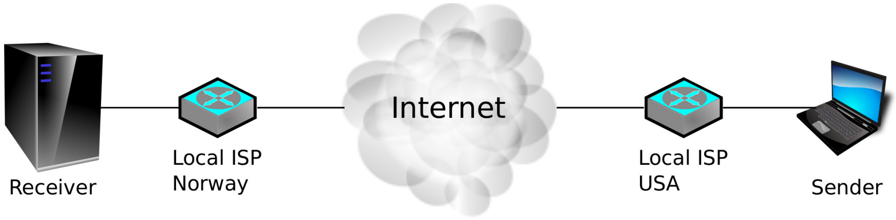


Figure 5.3: Test setup for Internet tests. Loss and delay are determined by path and competing traffic.

competing traffic. We can vary the packet IAT and size, but have no further influence over the network properties[5]. Traces of the sent traffic were gathered at sender and receiver (as for the laboratory tests) to be analysed.

5.1.5 Test data

One of the challenges when performing experiments to determine the effect of thin streams on reliable protocols is to generate the data patterns for the evaluated streams. For our tests, we have chosen two different approaches, each with its own purpose:

1. Traffic generating application: We have implemented applications that generate traffic according to special patterns. The packet IAT and size can be customised to reflect different thin stream scenarios. This approach is used to determine the effect of our modifications as the thin-stream properties vary.

2. Replay of traces: The traces that we analysed to identify the thin-stream properties are captured from real applications (see table 2.1). They reflect the patterns that such thin-stream applications display during ordinary use. To measure the effect of our modifications in a realistic scenario with varying traffic patterns, we replay such traces.

The combination of the two strategies for generating data patterns with the flexibility of laboratory tests and Internet testing provide a broad range of data for analysis.

5.1.6 Metrics

In order to measure the effect of our mechanisms, we had to decide on a set of metrics. The metrics were chosen to describe both the effect on latency, the consumption of network resources and the effect on competing streams.

- ACK latency: Describes the time interval from the first time a data segment is sent until the time when an ACK arrives to confirm that the segment is successfully delivered.

[5] The loss rate can be influenced by sending greedy streams over the same path at the time of the test. We have, however, not wanted to do that since we want the network characteristics to reflect normal scenarios.

- Delivery latency: Describes the time interval from the first time a data segment is sent until it is delivered at the receiver. The one-way-delay (OWD) is subtracted from the value, so that a packet delivered within one OWD has a delivery latency of 0 ms[6].
  - Transport layer: The delivery latency at the transport layer. Provides per-packet statistics.
  - Application layer: Delay from the time the data is sent until it is actually delivered to the application. Takes into account the in-order requirement.
- Loss: Loss rate for the connection that was evaluated.
- Transmission overhead (*tohd*): Measures the amount of overhead to successfully deliver the data in percent. We have to keep in mind that for thin streams, a large overhead (measured in percent) may constitute only a small amount of consumed bandwidth.
- Throughput: The aggregated throughput over a period of time is used to measure the impact of different retransmission mechanisms on per-stream fairness.
- Retransmission reason (SCTP): SCTP is a chunk-oriented protocol with a range of different strategies for bundling chunks. We have logged what triggered the retransmission to understand the dynamics of SCTP retransmissions better.

5.1.7 Loss estimation

In the cases where loss was not created using a network emulator, we needed to calculate or estimate the loss rate. In the cases where dumps were available both from the sender and receiver machines, loss was calculated as shown in definition 1. The numbers of sent and received packets were counted, and the difference accounted for lost packets.

Definition 1 (Loss estimation - sender and receiver dumps)

$$loss\ rate = \frac{sent\ packets - received\ packets}{sent\ packets} \tag{5.1}$$

Definition 2 (Loss estimation - sender dump only)

$$loss\ rate = \frac{Registered\ retransmitted\ packets}{Total\ number\ of\ sent\ packets} \tag{5.2}$$

[6] A detailed description of the delivery delay-metric and how it is calculated is provided in section 5.1.9

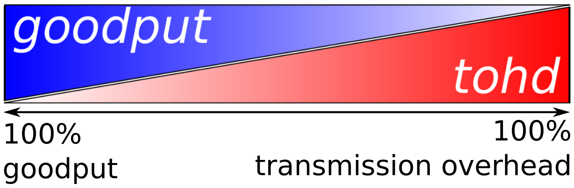


Figure 5.4: Relationship between transmission overhead and goodput in the statistics from the laboratory tests.

In the cases where we only had access to a sender-side dump, the loss rate was estimated from the number of retransmissions (as shown in definition 2). This way of determining loss is affected by the retransmission mechanisms, and also bundling skews this estimate severely. We have therefore avoided to use this way of measuring loss, only using it to confirm the loss rates (using TCP New Reno) in cases where we have no influence on the network behaviour. The method we used for calculating loss is specified for each presented test.

5.1.8 Calculation of transmission overhead

A metric was needed to analyse the transmission overhead (*tohd*) for different retransmission scenarios. Since bundling involves copying data between packets, measuring *tohd* as a function of sent and received packets would not give an accurate result. We therefore decided to measure *tohd* based on the number of sent bytes compared to optimal (lossless) delivery.

Definition 3 (Transmission overhead (*tohd*))

$$\omega = Outgoing\ bytes\ (aggregated\ payload)\ from\ sender$$

$$tohd = \frac{\omega - (seq_{max} - seq_{min})}{\omega}$$

Definition 3 shows our way of measuring the transmission overhead (*tohd*). We define *tohd* as the sent data in excess of perfect (lossless) delivery of all data as a percentage of the data size. Figure 5.4 shows how our metric works. In the case of 0% *tohd*, all the data is delivered with no need for retransmissions at all. In the case of 5% loss, if only one retransmission is needed each time a packet is lost, the *tohd* is 5%. If no data is delivered to the receiver, the *tohd* is 100%.

5.1.9 Calculation of delivery delay

The cumulative distribution function (CDF) plots show the delivery delay to the application at the receiver. Figure 5.5 shows how we record the timestamp t at the sender and t' at the receiver. Since achieving accurate results by synchronising the clocks of the hosts is very difficult, we

$$\begin{aligned}\Delta t_i &= t'_i - t_i\\ \Delta t_{min} &= \min_{1\leq i\leq n} \Delta t_i\\ \phi_i &= \Delta t_i - \Delta t_{min}\end{aligned}$$

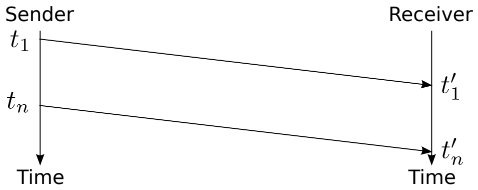


Figure 5.5: Calculating delivery delay (ϕ_i) by using timestamps at sender and receiver.

do not calculate absolute delay values. We assume that the lowest recorded difference Δt_i represents the OWD. For each sent segment, we find the delay by subtracting the OWD from the measured difference. Our recorded metric (ϕ) is thereby the delay above the lowest possible delivery time.

For tests that span several hours or days, the possibility that the clocks will drift is high[7]. We therefore find $\Delta t_{1,min}$ for the first 1000 packets, and $\Delta t_{2,min}$ for the last 1000 packets of the trace. We assume that both values represent the one-way delay at the given time, and calculate the drift τ as described in equation 5.3.

$$\begin{aligned}\Delta t_{1,min} &= \min_{1\leq i\leq 1000} \Delta t_i\\ \Delta t_{2,min} &= \min_{n-1000\leq i\leq n} \Delta t_i\\ \tau &= \Delta t_{2,min} - \Delta t_{1,min}\end{aligned} \tag{5.3}$$

The drift modifier τ is applied to each ϕ-measurement to compensate for the level of drift at the time the measurement was made. If the path (in an Internet test) is radically changed during an experiment, the change is clearly visible from the analysed data, and the test is discarded since no valid delivery delay can be calculated.

5.2 Evaluation of TCP modifications

We have performed a large number of experiments to measure the effect of the thin-stream modifications in different scenarios. First, laboratory tests were run in order to determine the explicit effect of each modification. The laboratory tests were performed with loss created by a network emulator, and with loss created by competing traffic. Experiments were then performed using the modified TCP to transmit over the Internet. They included a range of experiments to chart the effect of the thin-stream modifications on competing streams in order to address fairness. This section describes the experiments and results.

[7] All our measurements indicate that the drift is linear, meaning that we can compensate for the effect by calculating the rate of the drift.

RTT (ms)	50, 100, 150, 200, 250, 300
Loss	5%
IAT (ms)	25, 50 100, 150, 200, 250, 500
Packet size (B)	100
TCP version	New Reno, LT, mFR, LT + mFR, RDB, LT + mFR + RDB

Table 5.1: Laboratory test parameters for tests performed with uniform loss.

5.2.1 Artificial, uniform loss in an emulated network

The first part of our experiments evaluating the properties of our thin-stream TCP modifications was controlled laboratory tests to measure the effect of each modification. The laboratory setup and loss model used were as described in section 5.1.2 with uniformly distributed loss created by a network emulator. The duration of each test was 8 hours. Since we had implemented per-stream *IOcontrol* enabling of the modifications, we were able to run several streams in parallel during one 8 hour test period[8]. At the network emulator, the RTT was varied for different tests. The stream characteristics (packet IAT and size) were configured using our own data generator application *streamzero*. The parameters that were used for this set of tests are listed in table 5.1. All combinations of all the listed parameters were tested for a total of 294 different traces, each representing an 8 hour test. The TCP variations tested were: TCP New Reno, linear timeouts (LT), modified fast retransmit (mFR) and RDB. Combinations of our modifications were also tested as indicated in table 5.1.

As the test process was very time-consuming, we had to prioritise the parameters to vary. The choice was made to keep the packet size and loss rate constant for this test. The loss rate was set to 5% since we wanted to measure the effect when retransmissions happen, and therefore needed the loss rate to be relatively high. Effects when the loss rate is varied are described in more detail in the experiments in section 5.2.2 where loss is created by competing traffic and in the Internet experiments (section 5.2.4). The packet size was kept at a constant 100 bytes since we wanted the bundling mechanism to be active, and this is a packet size that reflects most scenarios as indicated by table 2.1.

Here, we present a subset of the test results that highlight interesting aspects of how each thin-stream modification performs under particular conditions. Graphs of the results for the whole series of tests can be found in appendix C.1.

The statistics for this test set are presented through three different kinds of plot: 1) A box plot representing the delay from the time when a data segment is sent until it is acknowledged. We call this interval the *ACK latency*. 2) A CDF showing the latency for delivery to the transport

[8] Using IOcontrols, we could turn on each mechanism separately for each stream. Using pre-defined port numbers for each sent stream we could transmit several streams in parallel, and still be able to analyse the results from each separate stream.

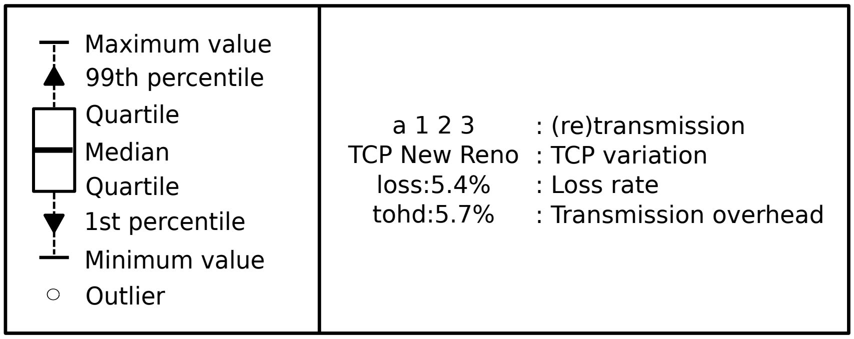


Figure 5.6: Legend for the boxplots used to describe the ACK latency for our TCP experiments.

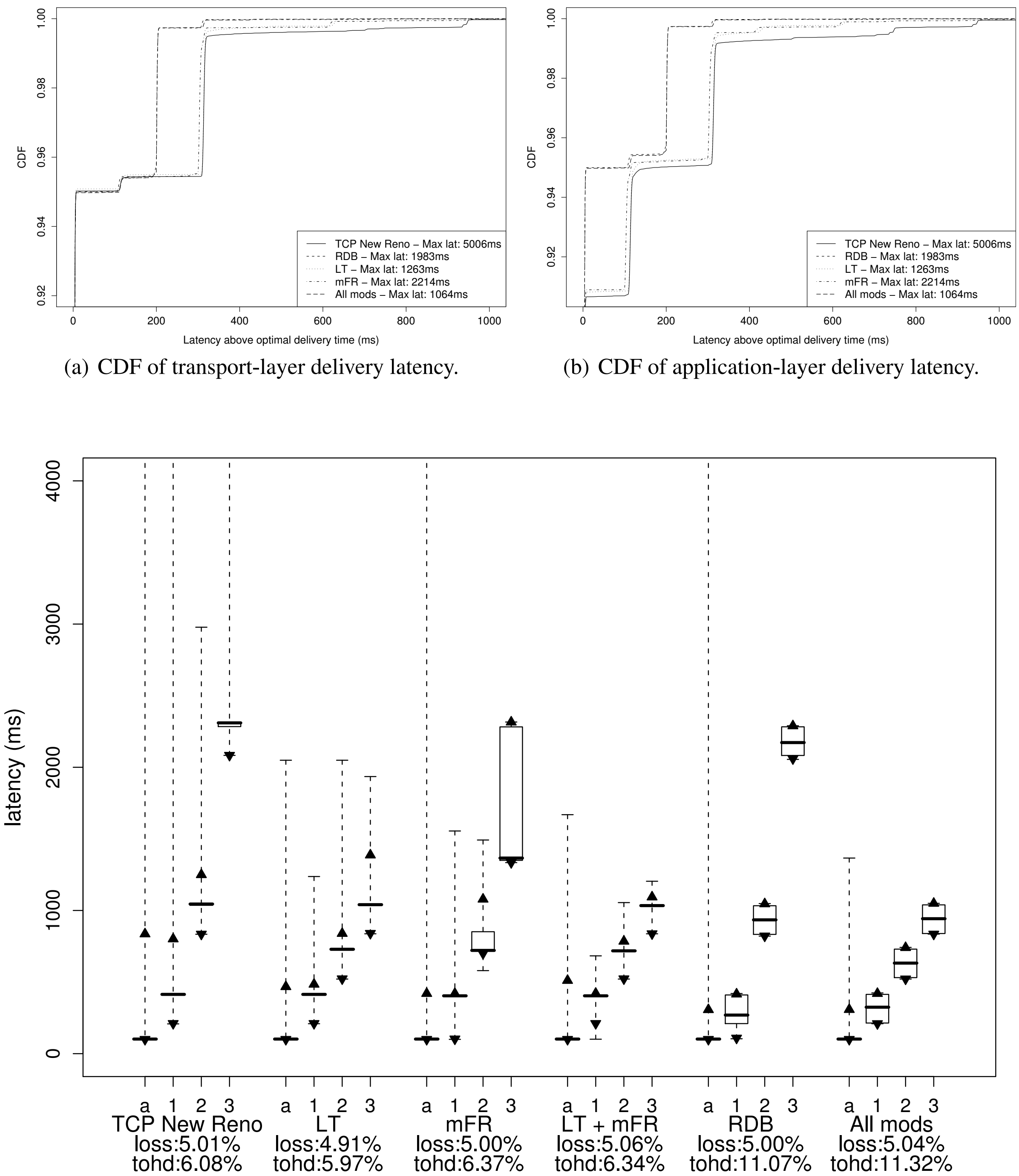
layer at the receiver. 3) A CDF showing the latency for delivery to the application layer at the receiver.

In the boxplot, each set of four boxes (“a, 1, 2, 3”) shows the ACK latency for one stream over an 8 hour test period. The item labelled “a” on the x-axis shows statistics for all packets, whereas “1”, “2” and “3” shows statistics for 1st, 2nd and 3rd retransmission. Figure 5.6 shows the legend for the box plots. The values printed beneath each item on the x-axis describe loss and *tohd* for the respective test. Loss is either estimated (from a server-side dump) or calculated from server and receiver dumps (as described in section 5.1.7). The *tohd* is a metric for the transmission overhead, and is calculated as described in section 5.1.8.

The CDFs show the share of successfully delivered data as a function of the delivery latency. At the point of the plot where the line reaches the number 1 on the y-axis, all data has been delivered. The value on the x-axis where the line reaches 1 on the y-axis reflects the highest delivery latency for the test. Since the maximum latency usually is very high for our tests, and the tested mechanisms differ only for a few percent of the transmitted segments, only the most relevant section of the total CDF is displayed. As the maximum delays of each test are relevant for the analysis, maximum values are shown as a part of the legend.

RTTs lower than IATs

Figure 5.7 shows the results for the test performed with an RTT of 100 ms and a packet IAT of 200 ms. The parameters reflect a typical thin stream with a packet IAT well above the path RTT. When studying the delays for unmodified TCP New Reno, we can clearly observe the exponential growth of retransmission delays in figure 5.7(c) as the number of retransmissions increases. In contrast to this, the LT section clearly reflects how avoiding the exponential backoffs helps reducing latency for such streams. In this test, the IAT is twice the RTT, which means that the chance for receiving three dupACK before a timeout is very slim. The mFR modification shows how the triggering of fast retransmit after the first dupACK improves latency for all recorded



(a) CDF of transport-layer delivery latency.

(b) CDF of application-layer delivery latency.

(c) Box plot of ACK latency

Figure 5.7: Latency statistics: RTT 100 ms, IAT 200 ms, packet size 100 bytes, uniform loss 5%.

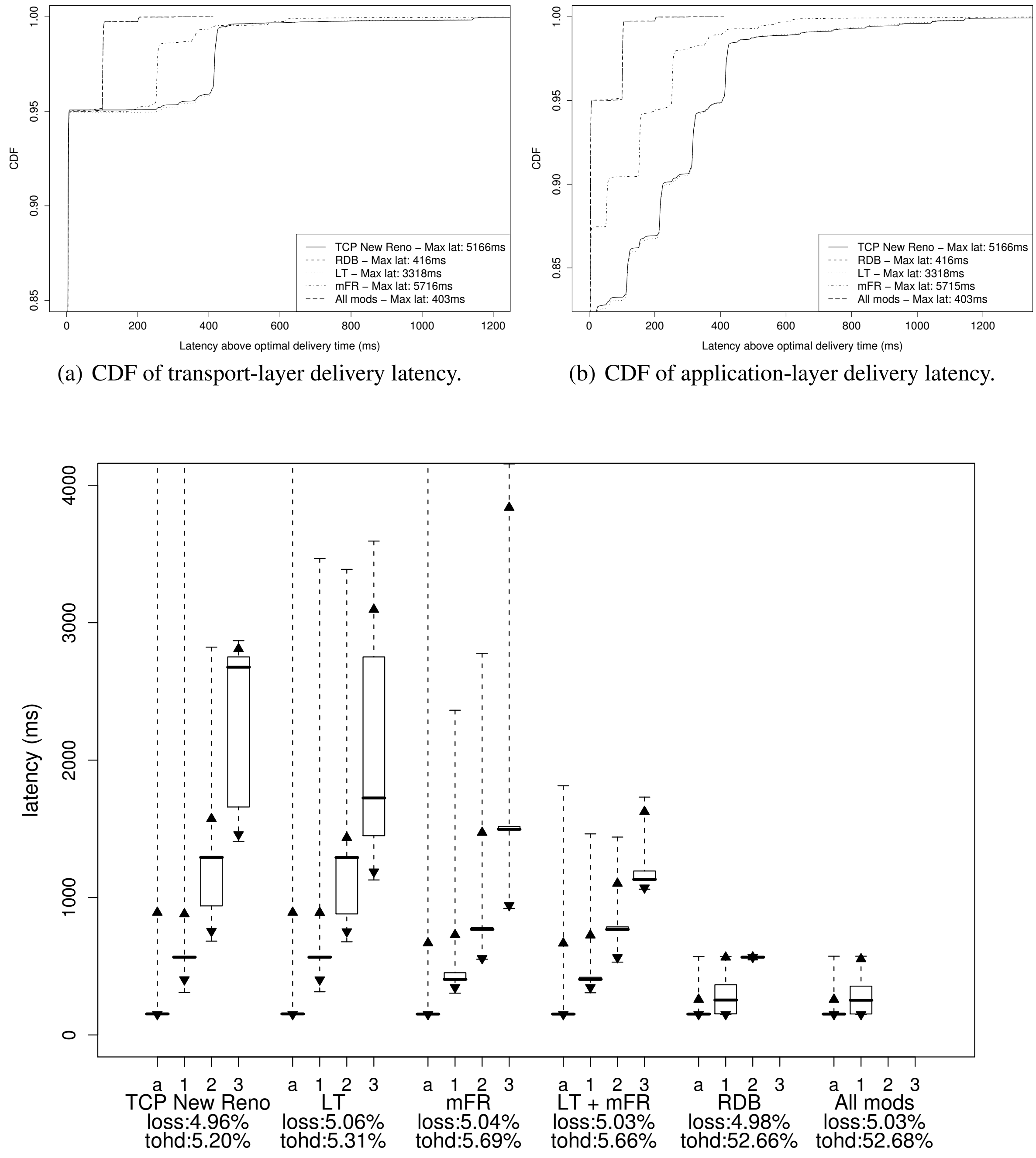
retransmissions. When we combine LT and mFR, we see that there is a general improvement, especially for the maximum and 99 percentile values. The *tohd* is slightly higher for the tests that apply the mFR modification. In this scenario, bundling does not happen for most transmissions (reflected in the relatively low *tohd* for RDB and All mods). Upon retransmission, though, bundling may be performed, giving a small decrease in latency for all retransmissions. When all the thin-stream modifications are combined, the result for this test is lowered latency with small variance for all retransmissions.

The results seen in the boxplot are also reflected in the CDFs of delivery latency shown in figures 5.7(a) and 5.7(b). We see that RDB and "all mods" are able to deliver most of the data earlier than the other tested options. The reason for this is that segments that are bundled help to avoid delays caused by the requirement for in-order delivery to the application layer. Loss may also, in some cases, be preempted even if the IAT is this high. On the transport layer (figure 5.7(a)), RDB performs better for the last 5%, reflecting the loss rate for the test. The CDF of the application layer delivery latency, however, shows that the in-order requirement keeps another 4% of the data waiting to be delivered. This creates additional delay for the streams without RDB enabled. When RDB is enabled, the application layer and transport layer delivery delay are nearly equal, as lost data is delivered with the next packet. The maximum values show the effect of the LT mechanism that reduces the effect of the highest delays. Here, the combination of all the mechanisms performs best, and reduces the maximum latency to $\frac{1}{5}$ of the maximum latency registered for TCP New Reno.

One aspect of the CDF is that a repetitive pattern of measured values with low variance results in CDFs with clearly defined “break points” where the different retransmission mechanisms are in effect (as observed in figure 5.7). When there is a larger variance in the observed values, the “corners” where latency is increased are less defined or “smoother”. A gentle curve can also be caused by a larger variance in the observed values. This effect usually happens when the packet IATs are sufficiently low to enable other retransmission mechanisms than timeouts.

RTTs larger than IATs

The test presented in figure 5.8 shows a scenario where the packet IAT is lower than the RTT. This increases the chance of triggering fast retransmits, and is also suited for bundling. Since most of the data segments are delivered successfully, the variance, when all sent segments are considered, is low. The variance increases when fast retransmit or bundling are active, which spreads the measured values. The fact that loss occurs both upstream and downstream also increases the variance of the measurements. There is, however, a noticeable improvement in latency also when considering all transmissions (’a’). The 99 percentile mark for TCP New Reno in figure 5.8(c) lies at slightly below 1000 ms. This value is noticeably lowered by the modifications. We can also observe the effect of exponential backoff for the retransmissions,



(a) CDF of transport-layer delivery latency.

(b) CDF of application-layer delivery latency.

(c) Box plot of ACK latency.

Figure 5.8: Latency statistics: RTT 150 ms, IAT 100 ms, packet size 100 bytes, uniform loss 5%.

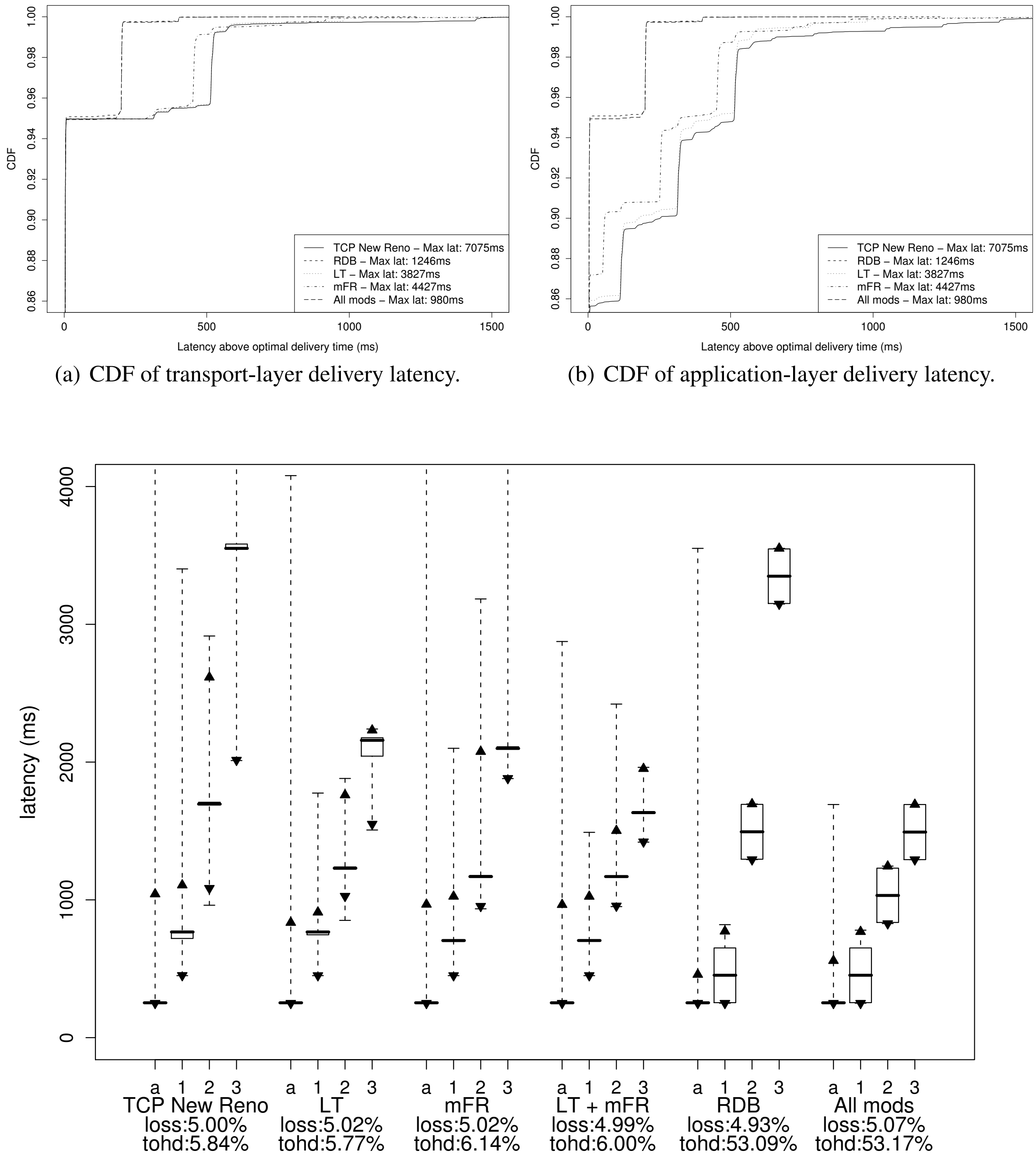
but not as pronounced as for figure 5.7, since fast retransmits are more likely to occur. The mFR mechanism makes a much larger impact than in figure 5.7, also caused by the lowered packet IAT. There are no occurrences of 2nd and 3rd retransmissions for the test with all modifications active. This is because RDB is able to deliver all the data before retransmissions are made. This comes at the *tohd* cost of 52.68% meaning that approximately one old segment was bundled with every new one.

The delivery latency on the transport layer (figure 5.8(a)) shows that RDB has generally the lowest latency. The mFR mechanism is able to lower delivery latency by almost 200 ms for the most delayed segments. LT and TCP New Reno show results that are very similar to each other. Figure 5.8(b) shows the CDF of the application layer delivery latency. Here, we see that RDB and the “All mods” tests perform best. In the middle, the mFR mechanism shows a reduction in latency compared to unmodified TCP and the LT mechanism that shows equal latency statistics. The “staircase” patterns in the CDF represent retransmissions, while the smaller variations and gentle slopes are caused by segments that have to wait for retransmission before they can be delivered to the application. This is also the reason why as much as 17% of the segments can be delayed even though the loss rate is 5%. The maximum values are significantly lowered when the modifications are active, mostly due to RDB for this scenario.

High IAT and RTT

In figure 5.9, the RTT and IAT are high. This scenario has the properties of an intercontinental connection. The high RTT is reflected in the RTO, which, in combination with the high IAT, makes reaction to loss slow. We can see the effect of this slow reaction in the high maximum value for TCP New Reno (over 7 seconds). In this scenario, the LT and mFR modifications show good results, as the IAT is high. RDB is able to bundle, and though the maximum latencies are high, the 99th percentiles for all transmissions are significantly lowered. The result of this bundling is a *tohd* of 53%. The *tohd* for LT and mFR is very close to that of TCP New Reno, indicating that the latency benefit of the LT and mFR modifications is very high compared to the “costs”.

The delivery latency on the transport layer seen in figure 5.9(a) shows properties that are very close to the one we saw for the previous test (figure 5.8(a)). This is a result of the IAT being lower than the RTT. The delays until recovery are higher, though, reflecting the high RTT of the connections. The application delay in figure 5.9(b) shows that a latency improvement is made by the LT modifications at each retransmission “step” of the ladder. The mFR modification gives an overall improvement whenever loss occurs, and RDB yields very good delivery latencies. The number of occurrences of 2nd and 3rd retransmissions is reduced by the preemptive effect of RDB, but when such retransmissions occasionally happen, the latency is not significantly improved. The combination of all mechanisms does, however, reduce de-



(a) CDF of transport-layer delivery latency.

(b) CDF of application-layer delivery latency.

(c) Box plot of ACK latency.

Figure 5.9: Latency statistics: RTT 250 ms, IAT 200 ms, packet size 100 bytes, uniform loss 5%.

livery latency significantly as shown in the CDF (figure 5.9(b)). The maximum values are also significantly lowered for all modifications.

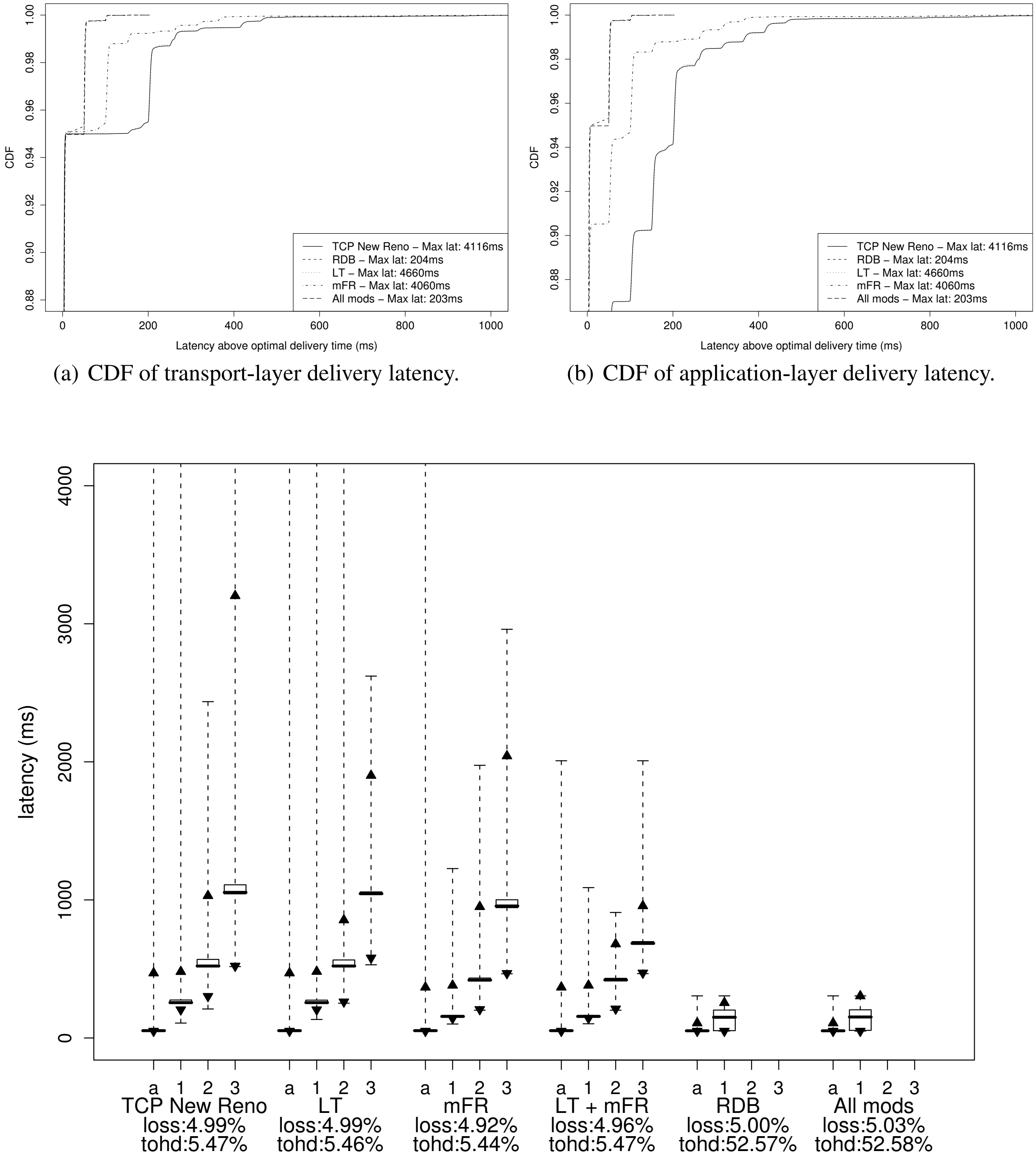
Low RTT and IAT

As an example of a lower RTT connection, we have chosen the test with an RTT of 50 ms and a packet IAT of 50 ms. Figure 5.10 shows the results for this experiment. Given the low RTT, feedback is usually received at the sender well before a timeout can be triggered. This is because of the 200 ms RTO$_{min}$. Even though the packet IAT equals the RTT in this scenario, the LT and mFR mechanisms show small effect. Fast retransmissions can be triggered before the RTO is triggered since the IAT is 50 ms, and four segments can be sent before the timer is triggered. The combination of the LT and mFR modifications reduces both the maximum ACK latency and the 99 percentile values, especially for the 2nd and 3rd retransmission. Bundling is possible, since the number of packets in transit is two on average for this scenario. Given the relatively high RTO$_{min}$ compared to the IAT, the bundled segments usually preempt the need for retransmissions, significantly reducing both ACK latency and delivery delay. The *tohd* for RDB in this test is 52.57% indicating an average of one bundled segment per transmission. The LT and mFR modifications display *tohd*-values that are very close to that of TCP New Reno.

The transport layer delivery shown in figure 5.10(a) shows that mFR is able to come close to RDB in delivery latency. TCP New Reno and the LT mechanism are close to each other in delivery delay. The maximum latencies for TCP New Reno, LT and mFR are also very close to each other, reflecting the regular TCP mechanisms' ability to provide lower latency in lower RTT / IAT scenarios. The positive effect of RDB on delivery latency for this scenario can be seen in figure 5.10(b). The maximum delivery latency when RDB is used is just above 200 ms. This is because retransmissions are almost totally avoided.

5.2.2 Congestion-Caused, variable loss in an emulated network

Uniform loss emulates some network scenarios, but very often loss patterns are bursty. The burstiness may increase latency because there is a greater probability that several retransmissions of the same segment are lost. Therefore, to generate loss by competition, we sent web traffic over the same emulated network, now with a bandwidth limitation and tail-dropping queue. Since the induced loss was generated by the emulated HTTP traffic, the average loss rate varied slightly from test to test. We also varied the intensity of the competing traffic in order to generate two levels of loss. Using these settings, we experienced an average packet loss of about 2% for a "low loss" set of tests, and 5% for a "high loss" set in the emulated network. All combinations of the parameters listed in table 5.2 were tested, and the setup is as described in section 5.1.3.



(a) CDF of transport-layer delivery latency.

(b) CDF of application-layer delivery latency.

(c) Box plot of ACK latency.

Figure 5.10: Latency statistics: RTT 50 ms, IAT 50 ms, packet size 100 bytes, uniform loss 5%.

RTT (ms)	50, 100, 150, 200, 250, 300
Loss	Cross-traffic 1-5%, cross-traffic 4-20%
IAT (ms)	25, 50, 100, 150, 200, 250, 500
Packet size (B)	100
TCP version	New Reno, LT, mFR, LT + mFR, RDB, LT + mFR + RDB

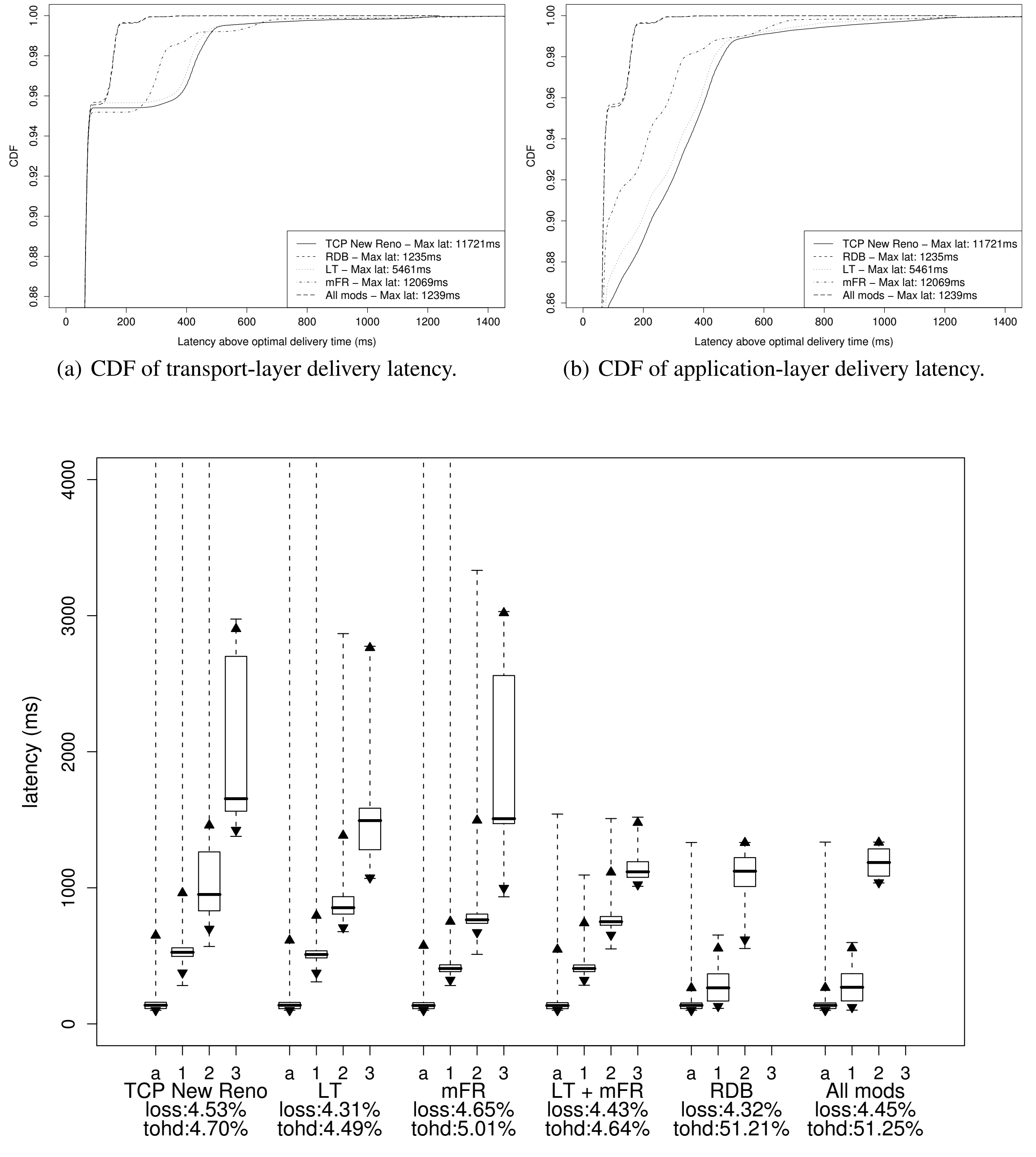
Table 5.2: Laboratory test parameters for tests performed with cross-traffic loss.

High loss

One of the defining factors of the cross-traffic tests is that the RTT of the test connection is correlated with the experienced loss rate. The generated cross traffic is set up with RTTs between 5 and 405 ms, equally distributed over 81 connections (see section 5.1.3). As a low-RTT connection gets feedback more quickly, it has a fairness-advantage over high-RTT connections that need more time to adapt to network conditions [74]. There are TCP variations that aim to even out this difference (like BIC and CUBIC), but since the thin-streams are nearly unaffected by *cwnd* dynamics, such schemes have shown small effects. The tests we have performed with high RTTs therefore show much higher loss rates than the tests performed with low RTTs, even though the cross-traffic parameters are identical.

Figure 5.11 shows cross-traffic tests with an RTT of 100 ms and a packet IAT of 100 ms. The loss rates shown in figure 5.11(c) are very close to those that we created with netem in the artificial loss-tests in section 5.2.1. An effect of the cross-traffic loss is that the variance is increased for all the analysed tests. Since the RTT is below the RTO$_{min}$, bundling and fast retransmit have an effect, even though the IAT is not lower than the RTT. We can see that LT alone does not have a large effect when compared to TCP New Reno. The mFR modification does not yield much improved latency when applied alone, either. When LT and mFR are combined, though, the effect is distinct, both for average, 99 percentiles and maximum values. The effect of bundling removes the need for more than two retransmissions for the analysed tests, and lowers the latency further. This is at the cost of redundancy, reflected in the *tohd* of 51.25%. The result when all the modifications are combined is very close to the RDB result, indicating that RDB is responsible for most of the gain seen for this test.

Figure 5.11(a) shows the transport layer delivery latency for this test. We can see that RDB and mFR have lower latency in the area where loss occurs. The application-layer effects of the observed improvements are made clear in figure 5.11(b). The lines in this CDF show more gentle slopes than what we have seen in section 5.2.1. This is caused by a larger variance in RTT and larger delivery delay due to the more bursty loss patterns generated by the cross-traffic. The delivery latency analysis also shows that mFR improves the perceived latency on the receiver side more than the box plot indicates. The LT mechanism yields a small improvement when compared to TCP New Reno. The main contribution for the LT mechanism here, is to reduce



(a) CDF of transport-layer delivery latency.

(b) CDF of application-layer delivery latency.

(c) Box plot of ACK latency.

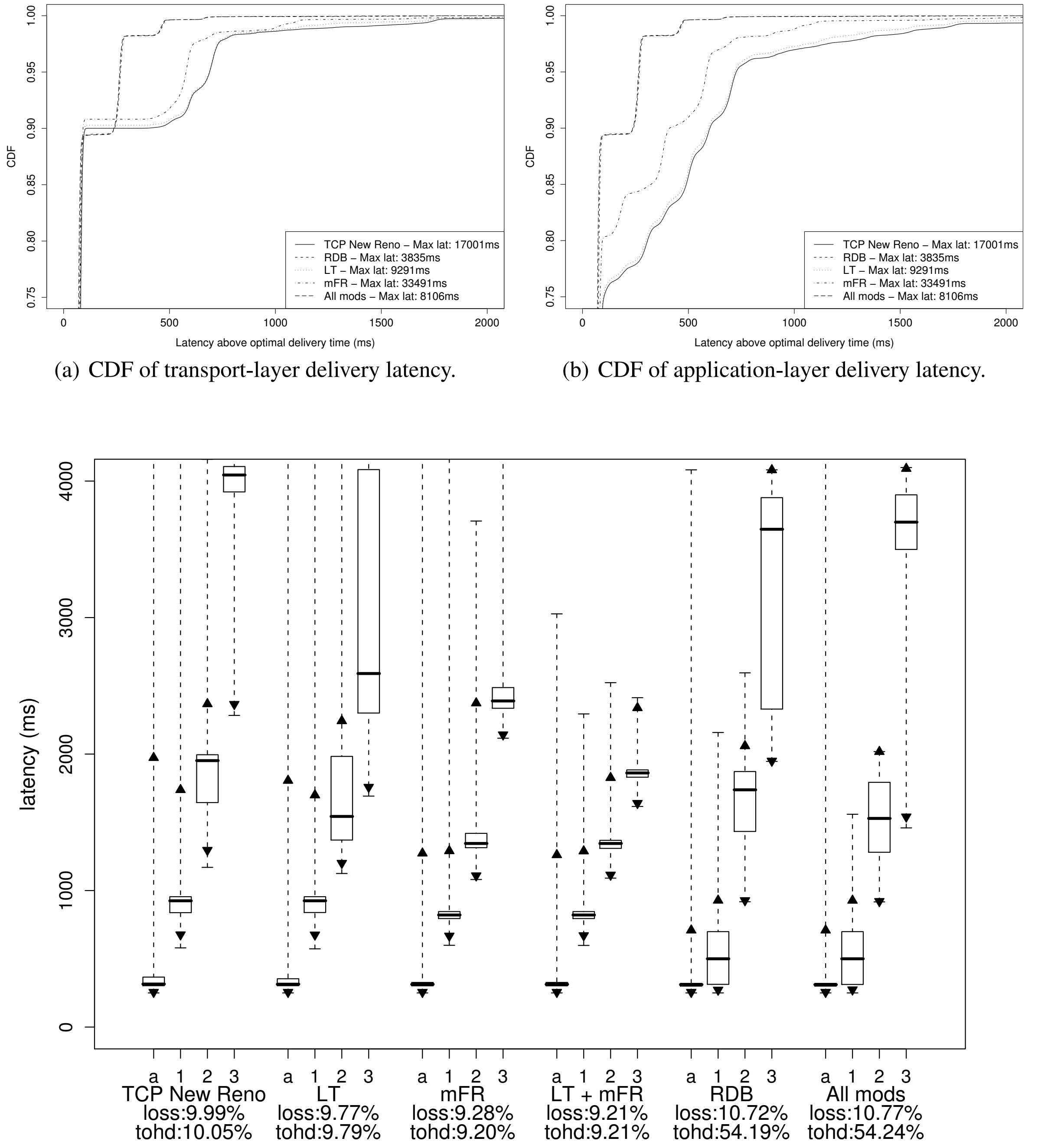
Figure 5.11: Latency statistics: RTT 100 ms, IAT 100 ms, packet size 100 bytes, cross-traffic loss. High loss rate.

the high maximum latencies that mFR and TCP New Reno show. It is clear that the LT and mFR mechanisms complement each other by providing lowered latency for different loss scenarios. The maximum latencies observed are generally higher for our cross-traffic experiments, even though the loss rate may be very close. This is probably also caused by the burstiness of the loss patterns caused by short periods of congestion over the bottleneck.

Figure 5.12 shows the statistics for the experiments performed with an RTT of 250 ms and an IAT of 200 ms. Because of the high test RTT, the analysed thin streams suffer very high loss in the competition over the bottleneck. As such, this test scenario reflects cases of extreme loss that are occasionally experienced on the Internet [92]. The boxplot in figure 5.12(c) shows how the extreme loss rate affects the ACK latency. Compared to TCP New Reno, the LT modification primarily reduces latency for the 2nd and 3rd retransmission. The IAT is lower than the RTT for this test, resulting in lowered latencies for mFR and RDB mechanisms. This effect can also be seen from the 99th percentile for all transmissions. For this test, the combination of LT and mFR give the lowest maximum latencies. The *tohd* for LT + mFR indicates that this gain is at a very low cost, as it is identical to the loss rate. RDB significantly improves the 99th percentile for all retransmissions, but at the relatively high cost of a 54.19% *tohd*. The results for all modifications are close to what is seen for RDB. The rather high latency experienced for 3rd retransmission in this test may be explained by a low level of samples for this test (only 9 samples of 3rd retransmission) due to the bundling performed. A somewhat higher loss rate also increases the chance for higher 3rd retransmission latencies.

In scenarios where the competition is this intense, small differences in the traffic can lead to congestion problems. For some cross-traffic tests we have noticed that the loss rate increases when RDB is active. This may be caused by a higher total number of sent packets when using RDB. Lost segments that are bundled and cumulatively ACKed mean that the stream does not detect the loss, and consequently does not back off exponentially. The absence of such long periods of waiting causes an increased number of sent packets for RDB.

Transport layer delivery latency for this test is shown in figure 5.12(a). The curves closely resemble the ones we saw from the uniform-loss test with same parameters (in figure 5.9(a)). Due to the increased loss rate, a larger percentage of the sent data is affected by increased delay. When comparing to the application layer latency in figure 5.12(b), we can see a gentler curve for TCP New Reno, LT and mFR than seen in the previous tests. The high loss rate increases the chance for several retransmissions of the same segment, causing higher delays for delivered segments due to the in-order requirement. The highest number of retransmissions for one segment when using TCP New Reno was 6 in this test. Waiting segments cause different delivery delays, explaining the gentle curves. The LT mechanism yields results that are very close to TCP New Reno, but we can see a lowered maximum latency. Improved latencies can be seen in the mFR statistics where, surprisingly, the maximum latency is highest by far. This



(a) CDF of transport-layer delivery latency.

(b) CDF of application-layer delivery latency.

(c) Box plot of ACK latency

Figure 5.12: Latency statistics: RTT 250 ms, IAT 200 ms, packet size 100 bytes, cross-traffic loss. High loss rate.

can be explained by the presence of exponential backoff and unfortunate coincidences (like the 6 subsequent retransmissions of the same segment). Both TCP New Reno and the mFR test have occurrences of 6 retransmissions. The high value of the maximum delay seen in the mFR test can be caused by lost ACKs that inflate the estimated RTT. RDB shows a significant improvement in latency, but at a cost in *tohd*. The same can be seen when all modifications are applied.

Low loss

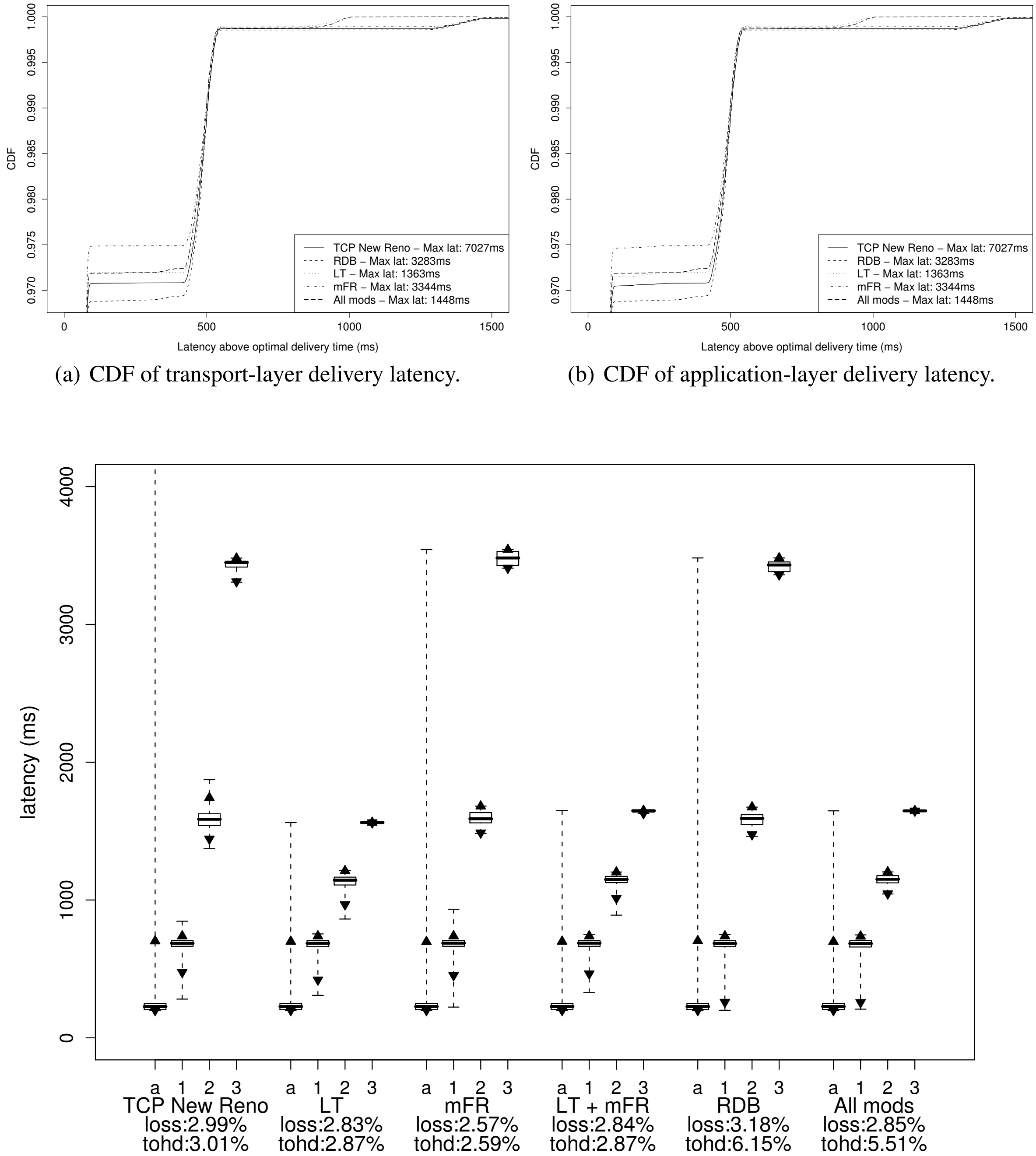
In the second configuration of the cross-traffic tests, the IAT of cross-traffic connections was reduced to keep loss well below 5%. This was done by increasing the IAT between each HTTP-like connection over the bottleneck. An important consideration was to keep loss low also for high RTT tests. The result is that loss is in the range of 1 to 3%. Such loss rates can often be experienced on the Internet, especially if a wireless link is used, or a gateway is shared by many connections. Even though the loss is lowered, many of the tests experience up to 6 retransmissions of the same segment, leading to high delays.

Figure 5.13 shows the results from tests performed with an RTT of 200 ms and a packet IAT of 500 ms. Even though the IAT is very high for this scenario, it reflects several of the thin-stream applications presented in chapter 2 (like SSH, Anarchy Online, CASA and Remote Desktop). For the boxplot shown in figure 5.13(c), the variance is very low for all modifications. This is caused by two main factors: 1) The high IAT means that almost all retransmissions are triggered by timeouts, 2) There are fewer samples due to the lower packet IAT.

Since most retransmissions are caused by timeouts, the exponential increase in latency for each new retransmission is very pronounced for TCP New Reno, mFR and RDB. The reason why RDB shows the same exponential curve is that bundles are not performed since two RTTs elapse between each transmitted packets. If a retransmitted packet is lost, the unacknowledged segment can be bundled. The high IAT also minimises the effect of the mFR mechanism. For this scenario, the LT mechanism helps reduce the severity of the latency events.

In figures 5.13(a) and 5.13(b), the differences between the modifications are only visible where the delay jumps from 50 to 450 ms. There are only minimal differences between transport and application layer latency. This is caused by the high packet IAT; every packet is delivered and ACKed before a new data segment is delivered from the application, even when loss occurs. An effect of the mFR mechanism can be observed, indicating that it leads to lower delivery latency in some cases. The LT mechanism and the combination of all the mechanisms show equal curves, which also points to LT being the most influential modifications when all are activated for this scenario. The maximum latency is reduced for all modifications compared to TCP new Reno.

With an RTT of 150 ms and an IAT of 50 ms, figure 5.14 shows a scenario typical for faster-

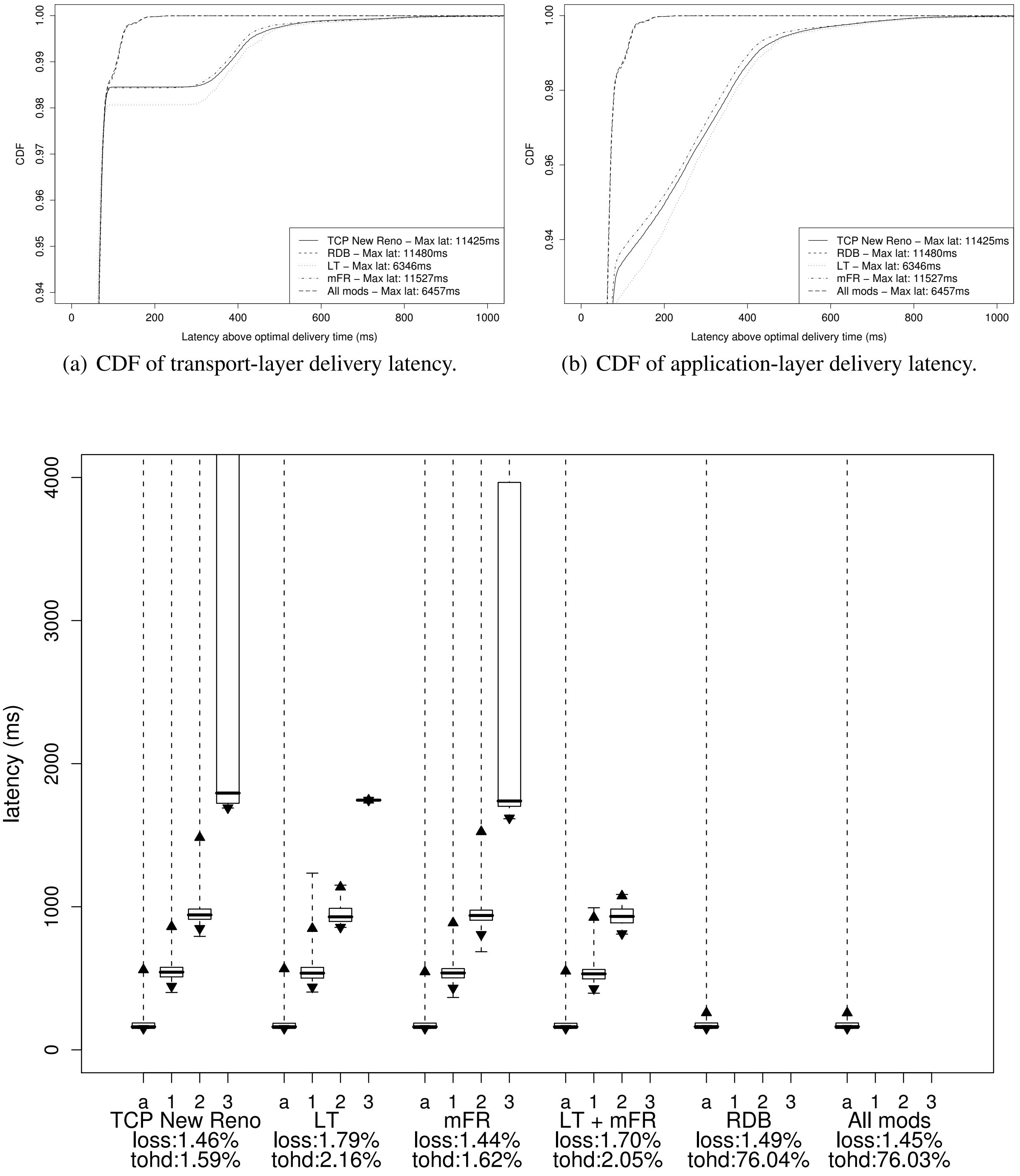


(a) CDF of transport-layer delivery latency.

(b) CDF of application-layer delivery latency.

(c) Box plot of ACK latency.

Figure 5.13: Latency statistics: RTT 200 ms, IAT 500 ms, packet size 100 bytes, cross-traffic loss. Low loss rate.



(a) CDF of transport-layer delivery latency.

(b) CDF of application-layer delivery latency.

(c) Box plot of ACK latency.

Figure 5.14: Latency statistics: RTT 150 ms, IAT 50 ms, packet size 100 bytes, cross-traffic loss. Low loss rate.

RTT (ms)	50, 150, 300
Loss	5%
IAT (ms)	25, 50, 100,150, 200, 250, 500
Packet size (B)	50, 100, 250, 500, 750, 1000, 1400
Bundle limit (B)	100, 400, 700, 1000, 1300, no limit

Table 5.3: Parameters for RDB limit laboratory experiments.

paced networked games, VNC or VoIP (see table 2.1). The relatively low IAT increases the effect on latency observed for RDB. Figure 5.14(c) shows the ACK latency statistics for this scenario. The LT mechanism reduces maximum latencies, and also affects 99 percentiles for the 2nd and 3rd retransmission. No significant effect can be seen from the mFR mechanism when studying the ACK latency. The delivery latency in figure 5.14(b), however, shows a small improvement. The combination of LT and mFR helps to keep most retransmission latencies well below 1000 ms. The difference in *tohd* between "LT + mFR" and TCP New Reno can be partly attributed to a somewhat larger loss rate in the "LT + mFR" test. A large latency improvement can be observed for the tests where RDB is active.

In figure 5.14(a), the RDB mechanism shows by far the best latency. Here, the LT mechanism actually performs worse than TCP New Reno. Figure 5.14(b) shows that the application layer latency for RDB is near-identical to the transport layer latency. The number of bundled segments, however, increases drastically, leading to a *tohd* of 76.04% for RDB. The segment size of 100 bytes leaves room for many segments to be bundled together. If the data segments delivered from the application were somewhat larger, the *tohd* would go down.

By presenting this subset of results from our laboratory experiments, we have tried to shed some light on the effect of each modification under different conditions. For complete statistics from this set of tests, consult appendices C and D.

5.2.3 Bundle-limit tests

A side effect of the bundling mechanism is the *tohd* that occurs as long as there is space (within the MTU) for bundling. Our tests show that, given a relatively low loss rate, a large gain in latency can be achieved by bundling only one or two segments. We wanted to determine whether an optimal ratio between *tohd* and achieved latency could be found. We performed experiments for measuring the effect of limited bundling in a controlled laboratory environment (as shown in section 5.1.2). Connection RTT, packet IAT, packet size and the maximum bytes allowed in a bundled packet were varied. All combinations of the parameters listed in table 5.3 were tested.

The plots displayed in this section is an analysis of the ACK latency from the tests. The loss rate and *tohd* are calculated as described in section 5.2.1. Inherent in the RDB mechanism

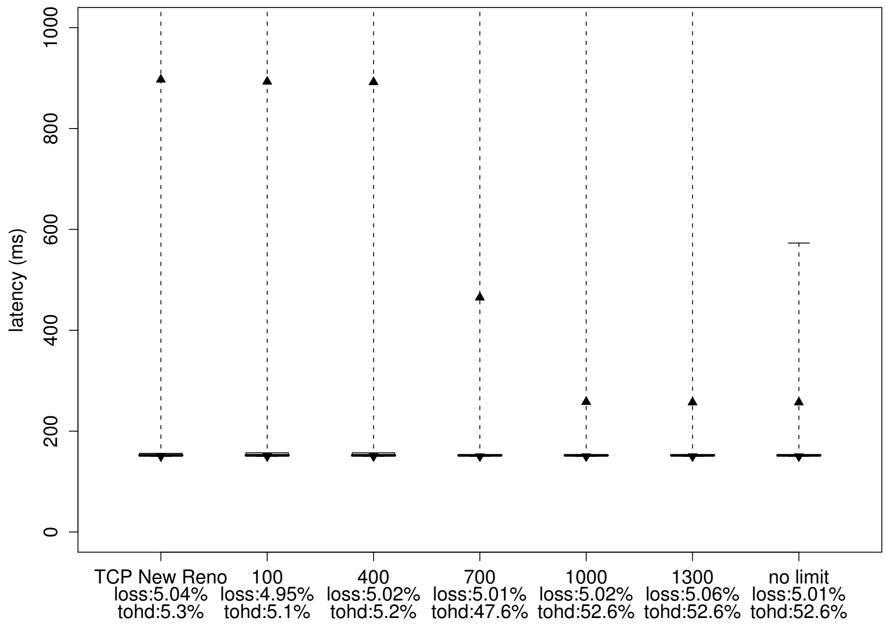


Figure 5.15: Varied RDB bundle limit - uniform loss - RTT 150, IAT 100, PS 250 Loss 5%.

is that bundling is possible when the $IAT < RTT$ and when retransmissions are performed. Our tests show that connections with high RTTs and relatively low IATs cause the *tohd* to soar (provided that the packet sizes are sufficiently small to enable bundling). Limiting the number of bytes that can be bundled reduces the *tohd* effectively. Unfortunately, as can be seen from our results, the latency is most reduced when the *tohd* is not limited. The mechanism is self-limiting, however, since bundling does not occur if: 1) The IAT becomes so low that TCP places the data segments in the same packet, and 2) TCP backs off because of loss and buffers data on the sender side, which results in full packets with no room for bundling.

A typical example of the effect of RDB can be seen in figure 5.15. In this test, the RTT is 150 ms, the IAT 200 ms and the packet size 250 bytes. When up to 500 bytes can be bundled, there will only be room for one segment, no bundling is performed and no effect is shown. When the limit is 700 bytes, a maximum of one extra segment can be bundled, and a reduction of the 99th percentile of the delivery latency is seen. From 1000 bytes limit and up, there seems to be enough room to bundle all waiting segments, and a large improvement of the 99th percentile is registered. As before, we can observe an increase in the *tohd* reflecting the average number of bundled segments.

There are examples where the bundling mechanism shows a very low *tohd*, but still achieves large latency benefits. Figure 5.16 shows the results from a test in which the RTT was relatively low (50 ms) and the IAT was 100 ms, a setup that is representative of a typical “thin stream”. In such cases, the sender receives feedback for each delivered segment before the application pro-

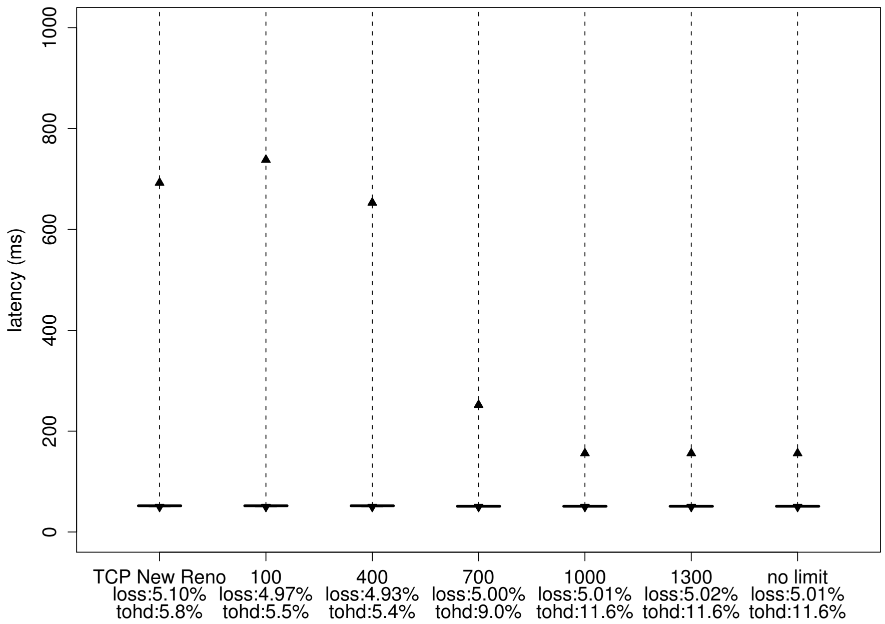


Figure 5.16: Varied RDB bundle limit - uniform loss - RTT 50, IAT 100, PS 250 Loss 5%.

duces a new segment to transmit. The result is that there is hardly any bundling. The exception occurs in the event of loss, in which case unacknowledged data is bundled with the next transmission. In figure 5.16, this can be seen by comparing the *tohd* as the bundle limit increases. We can see that for a loss rate of 5%, *tohd* never exceeds 12%. The general trend reflects the original assumption: a large improvement in latency can be achieved when several segments can be bundled, at the cost of increased *tohd*. A limit of one bundled segment yields a *tohd* of 50% and recovers data for cases where only one retransmission is needed for the segment. As more segments are bundled, subsequent losses of the same segment can be recovered. The bundle limit mechanism therefore seems best suited to avoid the most radical cases of *tohd* in scenarios where the packet IAT is relatively low and the packet sizes very small.

5.2.4 Internet tests

To determine whether our modifications also improve the latencies observed at the application layer in a real-world, Internet scenario, we replayed thin-stream traces (SSH, BZFlag and Anarchy Online) between a machine located in Worcester, Massachusetts (USA) and a machine in Oslo, Norway. Both machines were connected to commercial access networks. The results show that the proposed modifications generally improve the application-layer latency, which directly influences the user's QoE when retransmissions are necessary.

SSH

Figure 5.17 shows the delivery- and ACK latency for a SSH-dump replayed between access networks in Massachusetts, US and Oslo, Norway. Figure 5.17(a) shows the transport-layer delay, while figure 5.17(b) shows the application-layer delay. The lines in the plot show unmodified TCP and combinations of our modifications (RDB, LT, and mFR). The analysed streams were transmitted in parallel, resulting in comparable network conditions. The minimum latency observed for the SSH tests was 116 ms indicating that this the approximate path RTT.

For the SSH test, the loss rate was measured to ∼2%. On the transport layer (figure 5.17(a)), the difference between the tested TCP variants is made visible for the 2% with highest delivery latency. Unmodified TCP recovers most segments after 500-600 ms. RDB delivers some more data within the same latency interval, as do the retransmission modifications (LT + mFR). The combination of all mechanisms shows the best performance, but not by much.

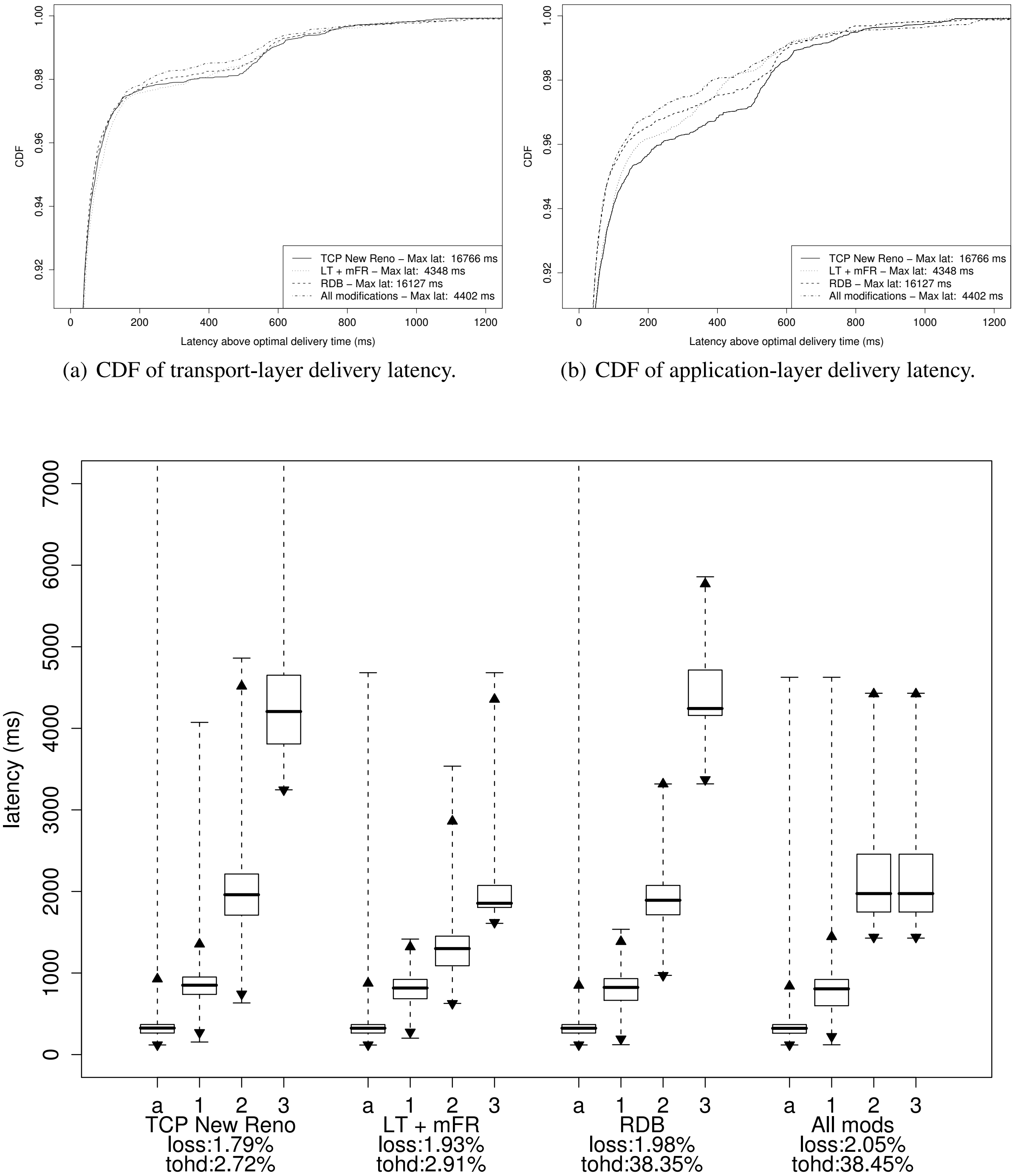
In contrast, when we study the results for the application-layer (figure 5.17(b)), we see that the seemingly small differences have a larger impact; the initial 2% loss now has latency consequences for more than 4% of the data.

We also see that the differences between the modifications are more pronounced and that the stream with all modifications active delivered 98.2% of the data at ∼500 ms, while unmodified TCP only delivered 96.6%. An important effect of the modifications is that they reduce the difference between transport-layer latency and application-layer latency, which is desirable for interactive applications.

The boxplot in figure 5.17(c) shows the ACK latencies for the same test. As the trace that was replayed has very high average packet IAT (see table 2.1), the possibility for bundling is limited. We can see that the effect from exponential backoff is present both for New Reno and for RDB. The combination of LT and mFR helps reduce the maximum latency to one fourth of the value observed for TCP New Reno (see maximum values in figure 5.17(b)). The limited possibility for bundling decreases the effect of RDB, this means that the *tohd* also remains low.

Anarchy Online

The Anarchy Online trace was replayed between the same endpoints as for the SSH-experiments (Worcester-Oslo). The observed path RTT was also the same as for the SSH-experiments: 116 ms. In figure 5.18, the results from this test are summarised. The loss rates we observed during this test were surprisingly high, 8.86% on average for the TCP New Reno stream. Figure 5.18(a) shows the transport layer delivery latency. On the transport layer, no difference between the TCP variations can be detected until 0.9 on the y-axis. This reflects the observed loss rate. RDB provides significantly lowered delivery latency for this experiment also. On the transport layer, 99% of the data is delivered within 1000 ms, while TCP New Reno delivers

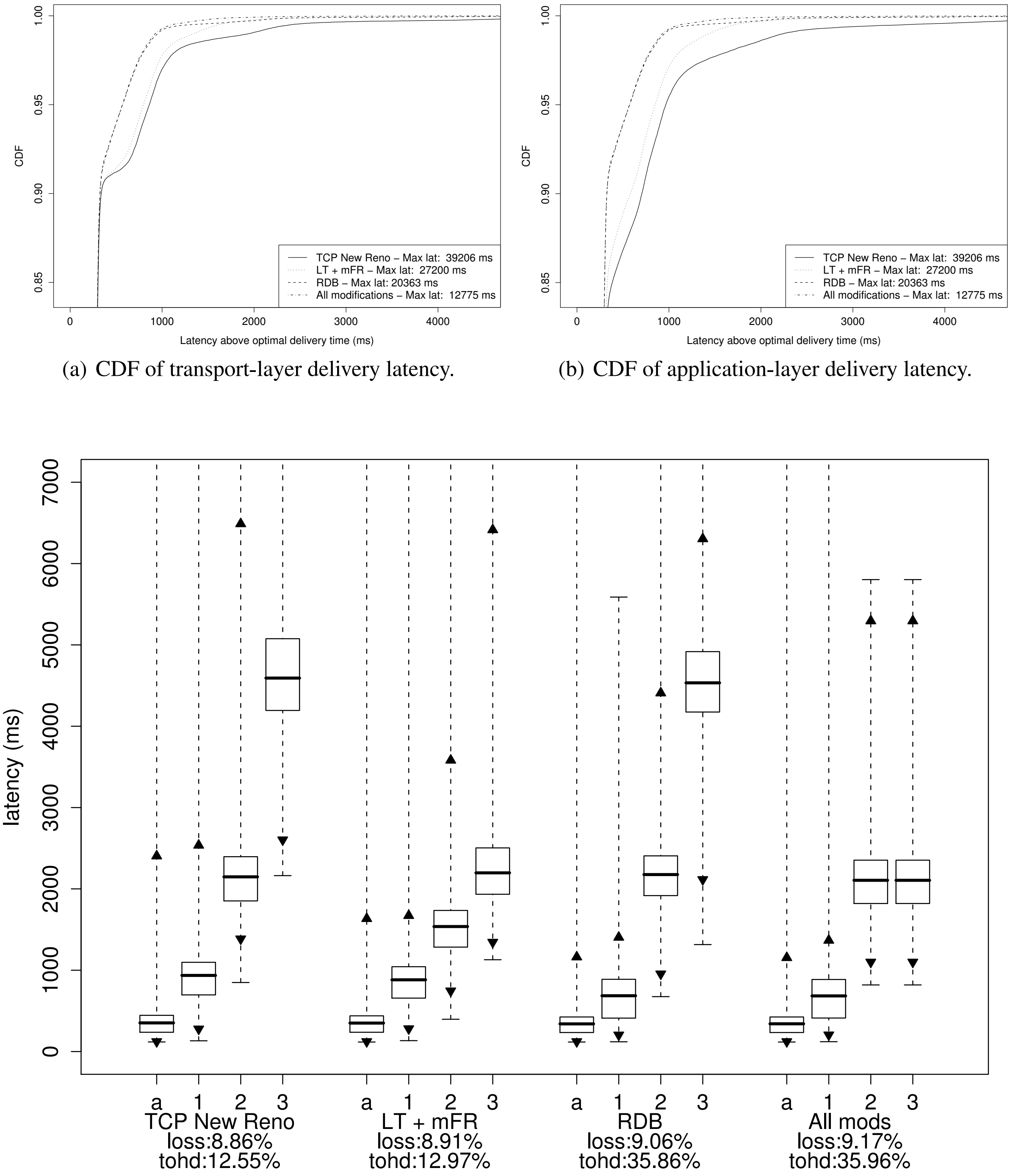


(a) CDF of transport-layer delivery latency.

(b) CDF of application-layer delivery latency.

(c) Box plot of ACK latency.

Figure 5.17: SSH trace replayed Massachusetts - Oslo.



(a) CDF of transport-layer delivery latency.

(b) CDF of application-layer delivery latency.

(c) Box plot of ACK latency.

Figure 5.18: Anarchy Online trace replayed Massachusetts - Oslo.

97% within 1000 ms. On the application layer (figure 5.18(b)), RDB still delivers 99% within 1000 ms, while for TCP New Reno almost 5% of the data is delayed more than 1000 ms. The LT + mFR mechanisms improve the latency somewhat, though not as much as RDB.

In the boxplot (figure 5.18(c)), TCP New Reno displays the expected exponential increase in latency for each retransmission. LT + mFR reduces this effect, while also reducing the 99th percentile value for all transmissions. RDB further reduces the 99th percentile. The maximum value is also reduced when applying the modifications, albeit still high due to the high loss rate.

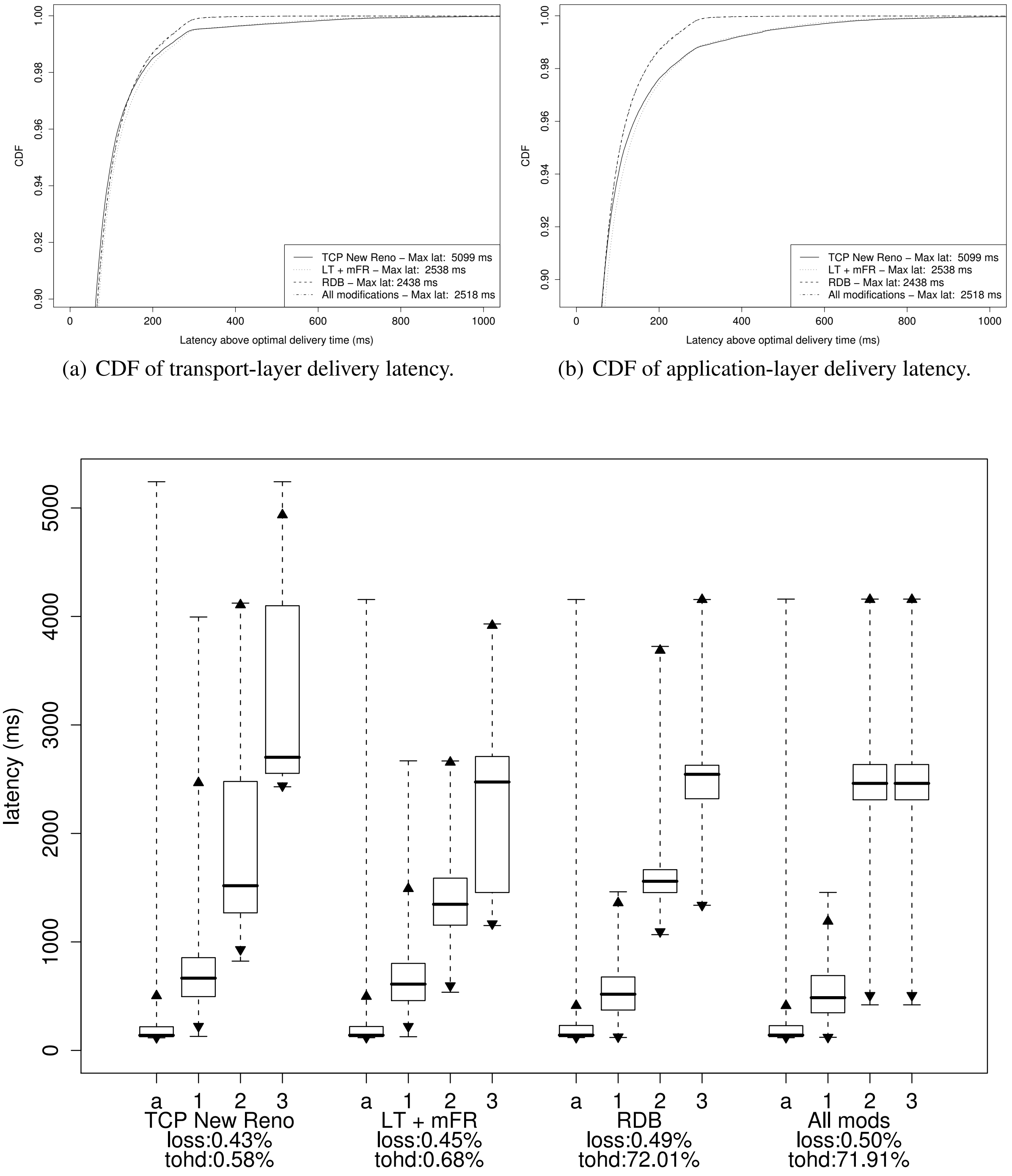
BZFlag

The third test performed over the Internet between Worcester and Oslo was the replayed trace from a BZFlag game (see table 2.1). Figure 5.19 shows the statistics from this experiment. The loss rate experienced when this test was run was much lower than for the previous two tests: around 0.5%. In figure 5.19(a), the transport layer delivery delay is shown. The gentle curve indicates that there is some variance in the measured RTO values. Only small differences between the TCP variations can be seen in this figure. For the 1% of the data with highest delivery latency, RDB shows somewhat better latencies than the other variations. On the application layer, this small difference has a larger effect. Here, the difference in delivery delay manifests itself for the last 4% of the data.

Figure 5.19(c) shows the ACK latency for the BZFlag test. The BZFlag trace has relatively low packet IATs (as shown in table 2.1). This increases the chance for fast retransmissions, but also makes bundling possible for most of the sent packets. The low packet IAT is reflected in the lowered 99th percentile and maximum value when RDB is active. The ideal bundling conditions result in a very high *tohd* of 72%. The combination of LT and mFR helps reduce the maximum latency. It also reduces the 99th percentile and maximum values for 2nd and 3rd retransmission. This is due to the mFR mechanism which ensures faster retransmissions for many lost segments due to the relatively low packet IAT for the BZFlag trace. The maximum values for TCP New Reno are halved when using the thin-stream mechanisms. This indicates that RDB shows an equal effect to the LT mechanism for lowering the maximum values when the packet IAT is relatively low.

5.2.5 Fairness

Our experiments and results demonstrate that the concept of improving latency by using a measure of redundancy is very promising from the viewpoint of the user of a thin-stream, interactive application. However, a measure of redundancy may mean that the streams using the modifications consume more of the network resources than they would normally do because we retransmit faster and fill each packet with more data. This might influence the performance of



(a) CDF of transport-layer delivery latency.

(b) CDF of application-layer delivery latency.

(c) Box plot of ACK latency.

Figure 5.19: BZFlag trace replayed Massachusetts - Oslo

other streams. That being so, we investigated how the proposed mechanisms influence other streams, greedy or thin, that compete for the resources on a common bottleneck.

The test setup for this set of experiments was like the one described in section 5.1.3. The modified streams were transmitted between one pair of computers, while the unmodified streams used the other pair. The network emulator was configured to create delay and bandwidth limitations. The bandwidth was limited to 1000kbps.

When many streams attempt to connect across a severely congested bottleneck, there are long connection setup delays. The three-way handshakes are not regulated by standard congestion control and affect the performance of the already connected streams. To avoid that connection attempts distort the throughput-analysis, we analysed the throughput of the connected streams only after all connections had been established successfully. By doing this, we were able to observe the effect of the congestion control mechanism dynamics without the interfering elements.

When a relatively low number of concurrent streams compete for the resources on the bottleneck, all the streams behave normally. As the number of concurrent streams increases, fairness breaks down given enough competing streams over a bottleneck[9]. The effect is that chance decides, to a larger degree, which streams are able to deliver data. This is caused by the massive loss experienced by all streams, which results in a constant state of recovery and slow start. If subsequent retransmissions of the same packet are lost, there are exponential backoffs whose duration is determined by RTO_{min} and the measured RTT. RTT measurements are very inaccurate in this case, because most segments are lost. A situation then arises in which retransmissions occur at intervals that are more or less random. None of the streams are able to back off (because they are already in the recovery phase), and the occasional retransmissions are sufficient to maintain the high level of congestion. Our measurements of such high-congestion scenarios indicate that this causes a situation in which one or more streams get significantly higher throughput than the others and that this occurs in a seemingly random manner. The described state of congestion-control breakdown occurs independent of whether our thin-stream modifications are activated or not. They are also independent of whether the streams are greedy or thin.

This situation of seeming randomness also occurs when a large number of thin streams (with or without the proposed modifications) compete for the resources over a bottleneck. The reason is that the thin streams never conform to regular congestion control, because they do not expand their congestion window; hence, they are not able to back off. This means that if the congestion is caused by a large enough number of streams (relative to the bottleneck capacity), the throughput is random and there is a sort of congestion collapse, from which none of the

[9] The number of concurrent streams that are needed for the described congestion breakdown is relative to the bottleneck capacity. For our tests, results started to get unpredictable when more than 256 concurrent streams were sharing the 1000 kbps bottleneck.

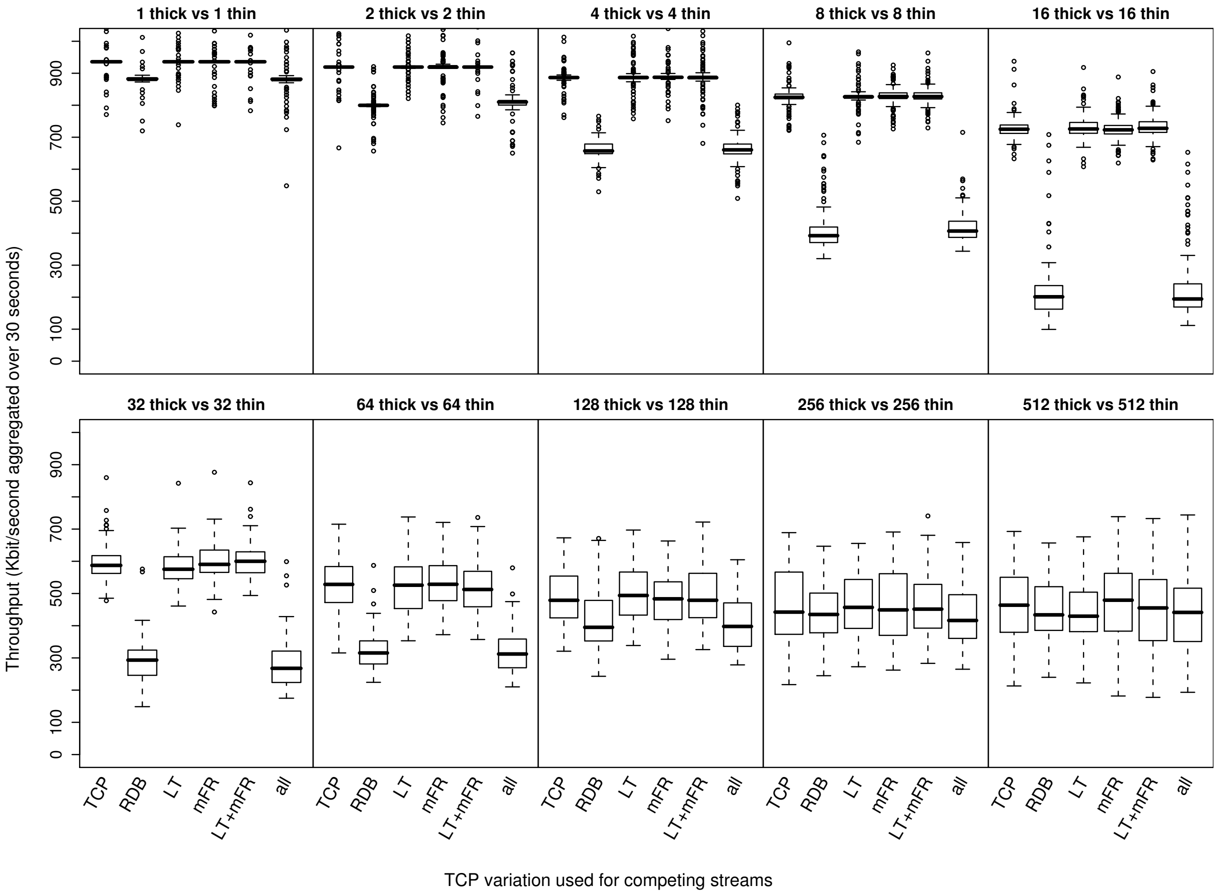


Figure 5.20: Aggregated throughput of *n* greedy unmodified TCP streams when competing with thin streams with different configurations/modifications. Thin-stream properties: Packet size: 120 B, Interarrival-time: 85 ms. Link properties: Bottleneck bandwidth: 1000 kbps, RTT: 100 ms.

streams can back off (because they are already in a state of a completely collapsed congestion window).

Figure 5.20 shows the results from a set of fairness tests, in which a set of greedy streams using standard (New Reno) TCP transmitted over a bottleneck with a rate limit of 1000 kbps. The greedy streams were exposed to competition from a set of thin streams, using a different configuration for each test. The boxplot shows statistics for the measured aggregated throughput for every 30 seconds for all greedy streams. The throughput was measured by analysing the acknowledgements (ACKs) on a sender side dump. The 30-second interval was chosen to avoid a lot of measurements where 0 bytes were ACKed (which would skew the statistics).

We can see from figure 5.20 that RDB grabs more bandwidth than the other TCP variants. The *tohd* for this combination of IAT and RTT when using RDB is ∼83%. As the number of competing thin streams increases, the effect from the competing RDB streams gradually increases until the greedy TCP streams are marginalised. We have to keep in mind, though, that each thin stream consumes much less bandwidth than a greedy stream would under the same conditions. From 64 competing streams and up, the differences in achieved throughput between streams that compete with RDB and streams that compete with other TCP variations narrow as loss forces the RDB streams to back off (sender-side buffering fills up the packets). We see that it becomes more difficult to find consistent patterns in the measured throughput when the competition reaches a point at which all streams go into (permanent) recovery (from $\sim$128 competing streams and up). For this scenario, the randomness surfaces regardless of the kind of modification the greedy streams is competing with. Another significant point from the analysis of figure 5.20 is that there are only very small differences in the measured throughput of the greedy streams when competing with the *LT*-mechanism or the *mFR*-mechanism (or both). This indicates that the relatively small overhead that is introduced upon retransmission for these mechanisms does not affect the throughput of competing streams for this scenario.

The streams are thin in the first place, with small packets and a low packet frequency, and therefore consume a very low amount of bandwidth. RDB shows a slightly higher number of sent packets than unmodified streams in lossy conditions because it receives more frequent feedback (from successfully delivered, bundled segments).

5.2.6 Comparison of thin stream performance in different operating systems

We have, in the course of our investigations, found very few available mechanisms that affect thin-stream latency positively in existing operating systems. However, operating systems are constantly modified and upgraded to adapt to new challenges. We therefore wanted to perform experiments with updated versions of some of the most commonly used operating systems to determine how they perform for thin-stream transfer, e.g.:

- Windows XP SP3: Windows XP is still one of the most used operating systems [18]. We therefore wanted to measure the performance of an updated version, namely Windows XP with service pack 3 and all available updates.
- Windows 7: Windows 7 is the up-and-coming operating system from Microsoft. It contains an updated network stack[10] and also implements a new TCP version: Compound TCP (CTCP) [72]. We have tested both the default Windows 7 TCP version and the new CTCP.
- FreeBSD 7.2: BSD is a common choice for servers and tailored systems. The BSD network stack has also traditionally been amongst the first to implement new networking solutions, and FreeBSD is often tuned for performance.

[10] The network stack was redesigned for Windows Vista [73].

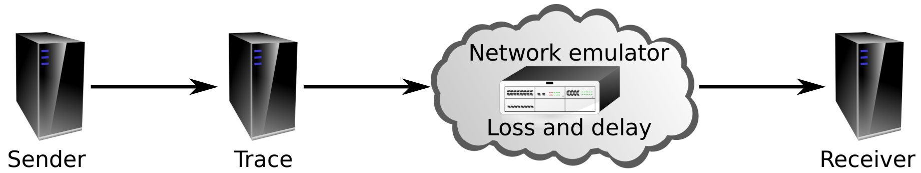


Figure 5.21: Test setup where a network emulator is used to create loss and delay. A separate machine is dedicated to create the packet trace.

- Linux (2.6.28): Linux is also a common choice for server environments. It is also gaining ground in the desktop segment and in handheld devices (i.e., through Android [47]).

When configuring each host for this set of experiments, we did our best to optimise each system for low network latency. This means that we disabled Nagle's algorithm where possible, and also searched for relevant options for each different system in order to get the best possible thin-stream latency. Another aspect of performing experiments on different operating systems is that traces may be affected by the respective systems' method for capturing data traffic. We therefore created a network setup based on the one described in section 5.1.2, but inserted an extra computer dedicated to creating the packet trace as shown in figure 5.21.

The test parameters were chosen to reflect a thin stream with a packet IAT that is greater than the RTT. The RTT was set to 150 ms, the packet IAT to 180 ms and the loss rate was set to 2% both upstream and downstream. The high packet IAT (compared to the RTT) means that Nagle's algorithm is not active in any case due to the lack of unacknowledged segments (see section 3.1.1)[11]. Using such a high packet IAT also assures that the traditional fast retransmission mechanisms is ineffective. To generate the transmitted data patterns, a Python-script was created that could be executed on all the tested operating systems.

The results from our experiments are shown in figure 5.22. Since only sender-side dumps were created for this set of experiments, the loss rates displayed are estimations based on the number of registered retransmissions (as described in section 5.1.7). This means that when bundles are performed (in "Linux mod"), it does not register as loss in the printed statistics since it preempts the retransmissions. The values for *tohd*, however, accurately picture the overhead calculated as described in section 5.1.8. For some of the traces, no occurrence of a 3rd retransmission was recorded. All the tested systems showed maximum values close to 2500 ms, except for FreeBSD and our modified Linux kernel. The 99th percentile for all transmissions is also significantly lower for Linux (2.6.28) and FreeBSD than for the tested Windows versions. The two tested TCP versions in Windows 7 show latencies that are very similar to each other (at least for all transmissions and 1st retransmission). The modified Linux version show lower

[11] This was a relevant consideration since we trusted in the IOcontrol-interface of Python to turn off Nagle's algorithm for Windows 7. This was done since we could not find any reference to registry entries that would force Nagle off (as for Windows XP).

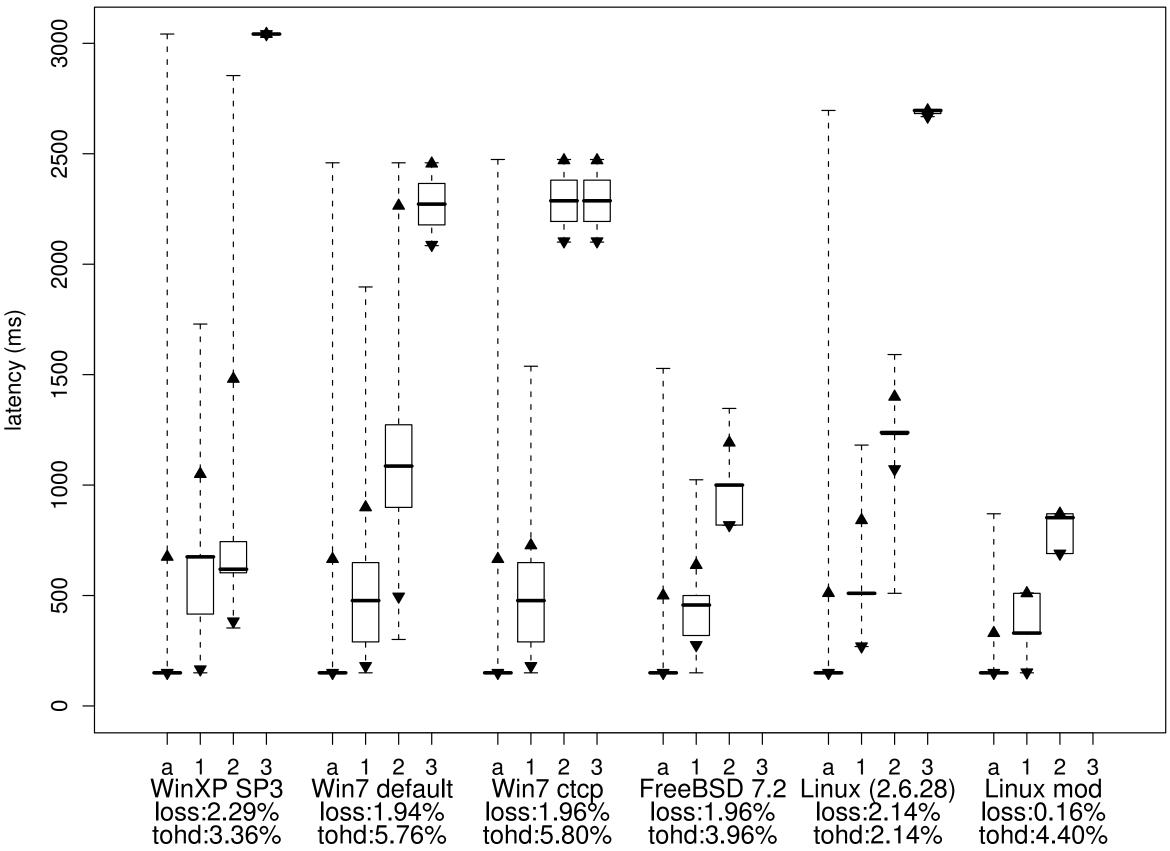


Figure 5.22: ACK latency for OS comparison. RTT= 150 ms, packet IAT=180 ms, loss=2%.

latency for all retransmissions than any of the other tested versions. The 99th percentile for all transmissions are also significantly lowered. When we look at the *tohd*, we can see that Linux (2.6.28) has a *tohd* of 2.14%, indicating that one retransmission was enough to recover the lost segment in most of the cases. Windows XP and FreeBSD show a little higher *tohd* with 3.36% and 3.96%, respectively. The modified Linux kernel gives a *tohd* of 4.40%, which is low, considering the increased aggressiveness of the mechanisms. The high IAT, however, limits bundling to cases where loss occurs (or extra delays happen). Surprisingly, the highest *tohd* is registered in the Windows 7 experiments with 5.76% for Windows 7 default and 5.80% for CTCP which implies aggressive behaviour for this scenario.

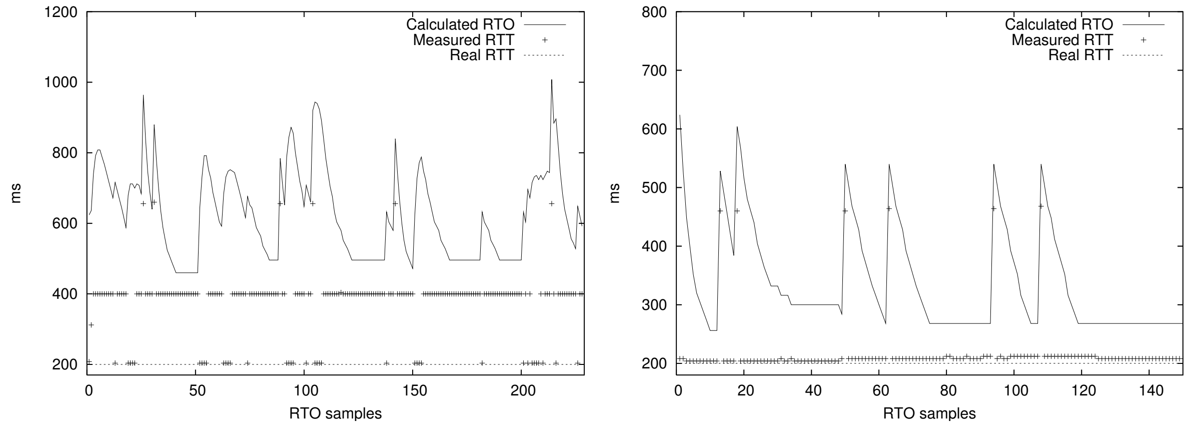
The performed tests indicate that the support for retransmission latency in thin-stream scenarios is poor, also in newer operating system versions. The improvement in the 99th percentile for all retransmission is approximately 200 ms to the second best result (FreeBSD). The maximum values are also significantly reduced.

5.2.7 Summary

In this section, we have shown the results of a wide range of experiments performed with thin-stream data traffic over TCP. Test were performed in the laboratory where we varied RTT and packet IAT, while testing the different modifications. We also performed experiments where loss was created by competing traffic. We experimented with different bundling limits when using RDB. Test were performed between access networks in Worcester, Massachusetts, US and Oslo, Norway, where packet traces from thin stream applications were replayed. To measure the effect of our modifications on competing traffic, we performed experiments on fairness. We also compared the thin-stream performance of TCP on different operating systems. From our tests, we can draw some conclusions on a general basis:

- Thin streams often produce high retransmission latencies when using TCP.
- When the IAT is low, mFR and RDB yield the largest latency-improvements. When the IAT is high, mFR and LT improve latencies, while RDB shows less effect.
- The modifications help to reduce latency both for low and high RTT connections.
- Improved latencies are seen when the modifications are active, even when the loss rates are very low. The improvements from our modifications are greater when the loss rate is higher.
- The maximum latency is often caused by multiple retransmissions by timeout, and is in many cases drastically reduced when using our modifications.
- We can detect only a small (or no) increase in *tohd* from the LT and mFR modifications. RDB can, in case of low packet IATs, result in high *tohd*.
- The difference between transport-layer and application-layer delays is significant when using TCP due to the in-order requirement for data delivery. Lowered retransmission delays can therefore result in large improvements in the user experience.
- The LT and mFR mechanisms do not affect throughput of competing streams enough to be detectable. RDB uses more resources than unmodified TCP on a congested bottleneck. When high loss rates makes streams back off, however, the effect is evened out.

In summary, our TCP experiments show reduced data delivery latency in thin-stream scenarios when our modifications are applied. Especially, the high worst-case delays that are damaging to the user experience are reduced. Fairness-experiments show that for our tested scenarios, the “cost” is usually negligible.



(a) Default configuration.

(b) Without delayed SACKs.

Figure 5.23: RTO calculated by SCTP with reduced RTO$_{min}$ for thin streams .

5.3 Evaluation of SCTP modifications

For evaluation of the modifications to SCTP described in section 4.2, a set of experiments were performed after the same patterns as for the TCP experiments (section 5.2). Both laboratory and Internet experiments were performed. In addition to using the laboratory experiments to determine the effect on thin-stream latency for different scenarios, we used chunk tagging to show accurately which mechanism triggered each retransmission. The effect on RTO calculation from delayed SACKs in *lksctp* was also analysed. Finally, we performed tests of throughput for two competing streams in order to give an indication of how the modifications affect fairness. The tests were performed on the 2.6.16 Linux kernel and FreeBSD 6.2. Note that the results from our SCTP experiments show the same trends as for TCP. We do therefore not include as many details since the conclusions are the same, i.e., our modifications improve the timeliness of data delivery for thin streams.

5.3.1 RTO calculation

As we performed the laboratory tests of SCTP, it soon became clear that delayed SACKs affect the RTO calculation to a large degree. This was made evident when we experimented with lowering the RTO$_{min}$ from one second to 200 ms (as is used by the Linux kernel TCP implementation). When the packet IATs are low, the frequency of feedback will be sufficient to achieve a reliable RTT estimation. For thin streams, however, the estimation is severely affected by the delayed SACKs. The effect of delayed SACKs on the RTO calculation can be seen from figure 5.23. Figure 5.23(a) shows how the calculated RTO fluctuates. The measured RTT is artificially high because of the delayed SACKS. When the RTT measurements occasionally reflect the actual RTT, the RTTVAR causes the RTO be inflated. Figure 5.23(b) shows the same

RTT (ms)	0, 50, 100, 200, 250, 400
Loss	1%, 5%
IAT (ms)	50, 100, 150, 200, 250
Packet size (B)	120
SCTP version	lksctp (unmodified), LT, minRTO, mFR, All modifications
TCP version	New Reno

Table 5.4: Laboratory test parameters for tests performed on SCTP with uniform loss.

tests without delayed SACKS. We can see peaks in the RTO measurement caused by sporadic high RTT measurements. The peaks occur much less frequently, though, and the peaks keep much lower values. The RTO peaks almost always stay below 1 second, which means that the RTO is normally rounded up to the RTO$_{min}$. When we perform experiments with a lowered RTO$_{min}$, however, the delayed SACKs result in higher retransmission latencies for the thin streams. Since the delayed SACKs make it difficult to compare the effect of our modifications to TCP, and since it increases latency for thin streams, delayed SACKs are turned off for our experiments.

5.3.2 Artificial, uniform loss in an emulated network

Laboratory experiments were set up to determine the effect of the SCTP modifications under different conditions. The experimental setup was as described in section 5.1.2. We wanted SCTP to be able to bundle using its intrinsic bundling schemes, so the packet size was set to a constant 120 bytes. The mechanisms we evaluated were: unmodified lksctp (lksctp), Linear timeouts (LT), modified RTO$_{min}$ (minRTO), modified fast retransmit (mFR) and all the modifications combined (all mods). The timer reset modification was active for all SCTP variations except unmodified lksctp. Earlier experiments [83] showed that bundling on fast retransmit did not enhance latency for thin streams, so this modification was not tested. Table 5.4 shows the test parameters for this group of tests. We used TCP New Reno as reference, since we have shown that this is the standard TCP version that shows the best thin-stream latencies [50]. Each test had a duration of 2 hours and was repeated several times to produce enough retransmissions to be statistically viable. In this test set, SCTP traffic was sent over an emulated network, introducing artificial loss and delays. As a representative example, we present the results of comparing lksctp with our modifications in a thin stream scenario. The tests produced a large number of 1st and 2nd retransmissions. The number of 3rd retransmissions is so low that it has to be regarded as an indication rather than a statistically viable dataset.

The SCTP tests showed results similar to the TCP tests (presented in section 5.2) when the RTT and IAT were varied. Figure 5.24 shows results from the SCTP tests when the RTT is 100 ms and the packet IAT is 250 ms. This is the typical thin-stream scenario where the packet

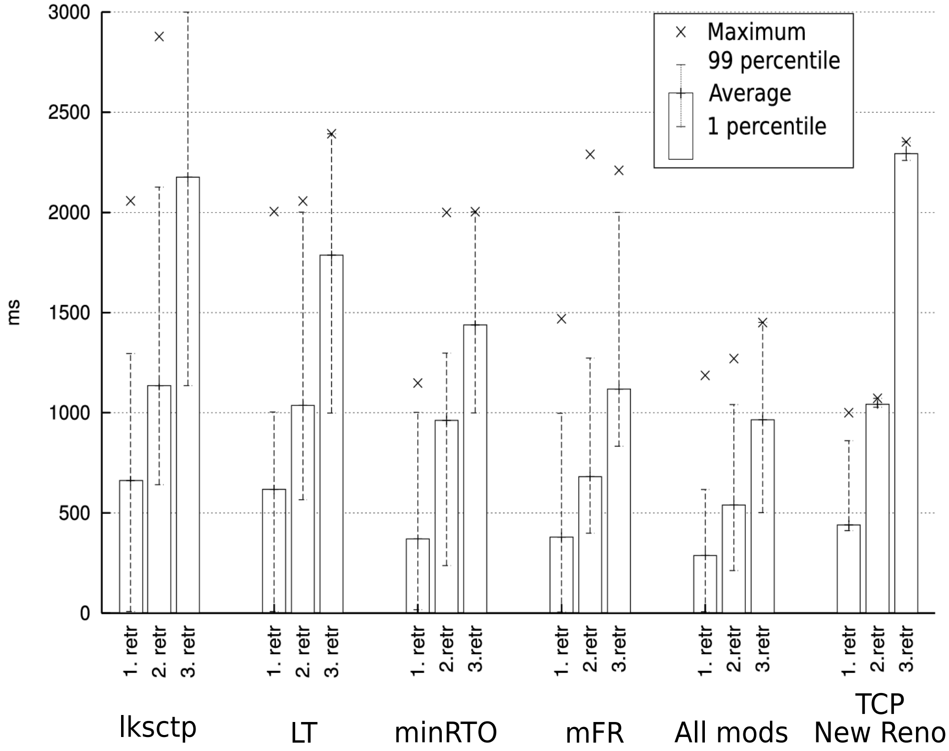


Figure 5.24: ACK latency. RTT=100 ms, packet IAT=250 ms. Uniform loss (5%).

IAT is larger than the connection RTT. When linear timeouts were used, we observed a reduction in maximum latencies compared to lksctp, especially for 2nd and 3rd retransmission. The 99th percentile and average latencies were only marginally reduced. With an RTO$_{min}$ of 200 ms, we saw improved average and 99-percentile latencies as well. The results can be explained by the fact that most retransmissions in thin stream scenarios are caused by timeouts. By reducing the RTO, the latencies for all retransmissions by timeout have been lowered. For the mFR mechanism, we saw that the average and 99-percentile latencies were drastically improved compared to lksctp. Maximum values were still high, caused by exponential backoff. The test where all our modifications were combined showed large improvements for maximum, 99-percentile and average latencies. Generally, we saw that improvements from the modifications got more pronounced on the 2nd and 3rd retransmission.

For the 1st and 2nd retransmission, TCP New Reno performs better than unmodified lksctp. On the 3rd retransmission, lksctp has a better average value, although the 99 percentiles and maximum latency are still better with TCP. However, our modified lksctp performs better than TCP except for maximum values of the 1st and 2nd retransmission. In the 3rd retransmission, however, TCP displays much higher maximum latencies than the modified lksctp. The difference between All mods and TCP New Reno is not very large for the 1st retransmission, but gradually increase as TCP New Reno backs off exponentially.

RTT (ms)	0, 50, 100, 200, 250, 400
Loss	Cross-traffic ~5%
IAT (ms)	50, 100, 150, 200, 250
Packet size (B)	120
SCTP version	lksctp (unmodified), LT, minRTO, mFR, All modifications
TCP version	New Reno

Table 5.5: Laboratory test parameters for tests performed on SCTP with congestion induced loss.

5.3.3 Congestion-caused, variable loss in an emulated network

Uniform loss emulates some network scenarios, but situations where the loss patterns are bursty are common on the Internet. Bursty loss patterns increase the chance that several retransmissions of the same chunk is lost. Therefore, to compete for resources with a more realistic load, we sent web traffic over the same emulated network to introduce congestion and thereby loss. The laboratory setup was as described in section 5.1.3. Since the induced loss was generated by the emulated HTTP traffic, the total loss varied slightly from test to test. On the bottleneck, the bandwidth was limited to 10Mbps with a queue length of 100 packets. Using these settings, we experienced an average packet loss of about 5% in the emulated network. Table 5.5 shows the test parameters for this group of tests. The cross-traffic induced loss makes for more bursty loss patterns than the uniform (netem) loss, also resulting in jitter as packets have to wait in the queue.

Figure 5.25 shows the ACK latency when loss is created by cross-traffic over a bottleneck. The results show the same patterns as for the test where the loss was uniformly distributed (figure 5.24). Unmodified lksctp shows large delays for all retransmissions, and displays the typical exponential increase of delays between subsequent retransmissions. When the LT modification is applied, maximum values are reduced. A larger reduction in average retransmission delays can be seen for the reduced RTO$_{min}$. For this scenario where the packet IAT is high, the mFR mechanism reduces latency, but not by much. The combination of all the mechanisms, however, results in a large reduction in retransmission latency for this test. The fact that figure 5.25 and figure 5.24 show so similar results indicates that the burstiness in loss patterns does not affect thin stream latency to a large degree. The effect of the different mechanisms, as packet IAT and RTT are varied, reflects what we have seen for the TCP tests in section 5.2. Lower IATs reduce latency when the mFR mechanism is active, and LT reduces maximum latency. The largest difference is the modified RTO$_{min}$, which reduces SCTP latency for all thin-stream scenarios.

5.3.4 Analysis of which mechanisms triggers retransmissions

SCTP can order data in a range of different ways due to its chunk-orientedness. This makes it difficult to determine the mechanism that triggers a retransmission of a given chunk. Such

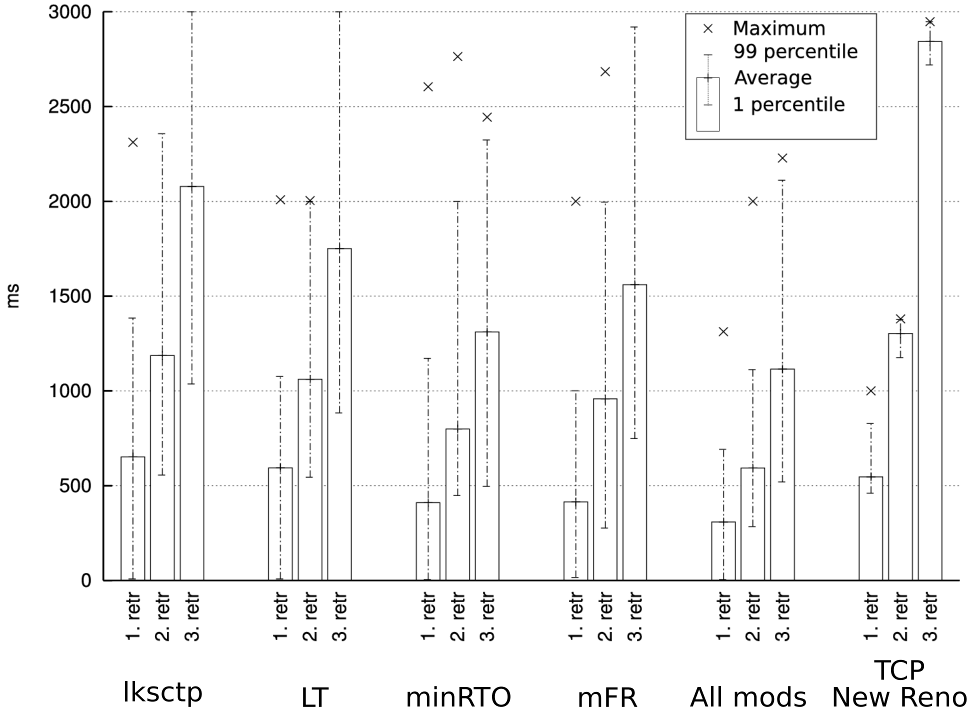
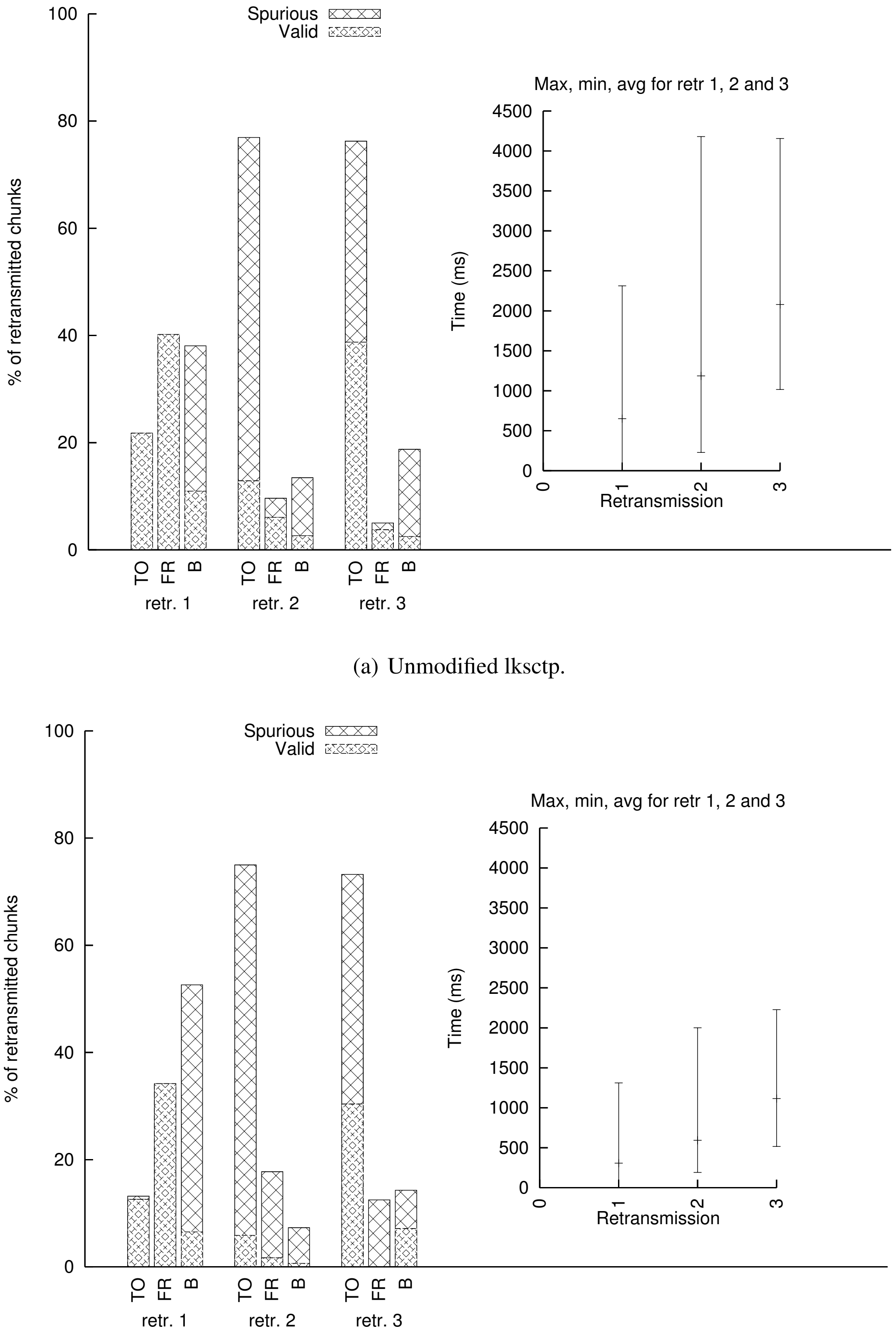


Figure 5.25: ACK latency. RTT= 100 ms, packet IAT=250 ms. Cross-traffic induced loss (5%).

knowledge about retransmissions can help us to understand the practical workings of the retransmission mechanisms better. We therefore made a system that allows us to identify the mechanism that retransmitted each chunk. To evaluate the effect of our modifications and to determine the extent of SCTP bundling, we created three "groups" of retransmissions for the purpose of this analysis: 1) Retransmissions by timeout, 2) Fast retransmissions and 3) "retransmissions" by bundling. The different bundling schemes employed by SCTP are all categorised into the same group. For this analysis, we modified the lksctp implementation to tag the payload of the chunk with an identifier, depending on the reason for retransmission (see section 4.2.1). This destroys the payload, but for this set of experiments, the payload is not relevant. We also sort the retransmissions into groups based on whether they are spurious or valid.

Figure 5.26(a) shows the reasons for retransmissions from a test where the RTT is 100 ms, the packet IAT is 250 ms and the loss rate is 5%. In this test, unmodified lksctp is used. We can see that fast retransmission is the dominant trigger for first retransmissions. Bundled chunks are the second most common source of retransmissions, but the majority of these are spurious. Timeouts represent a little more than 20%. For second retransmissions, the share of retransmissions due to timeouts increases. Timeouts are responsible for 75% of the retransmissions, and most of these are spurious. A reason for this may be that a timeout is triggered just before a SACK that acknowledges the retransmitted chunk is received. The timer reset (described in sec-



(a) Unmodified lksctp.

(b) All SCTP modifications.

Figure 5.26: Cause for retransmissions. RTT=100 ms, IAT=250 ms, loss=5%. The bar denoted *TO* represents timeouts, *FR* represents fast retransmissions, and *B* are bundled chunks.

tion 3.3.3) contributes to the early retransmissions. Most of the retransmissions by bundling are spurious also for the second retransmissions. Bundles constitute no more than 15% of the 2nd retransmissions, though. Although the number of samples for the third retransmission is low, the data indicate that timeouts still dominate. The relative share of spurious retransmissions is lower than for the second retransmissions. Here also, the majority of the triggered retransmissions is from timeouts. The ACK latency displayed in figure 5.26(a) shows that the maximum latency is high, even though there are many spurious retransmissions. This indicates that the spurious retransmissions do not help to reduce latency for this thin-stream experiment[12].

We compare this with the analysis of retransmissions for SCTP with all our thin-stream modifications active, shown in figure 5.26(b). For first retransmissions, the relative share of timeouts has been reduced. The share of fast retransmissions has also gone down by 5-6%. Bundled chunks represent a larger share than for unmodified lksctp, and the majority of these bundled chunks represent spurious retransmissions. This increase in the share of spurious bundled chunks indicates that the lowered RTO_{min} and modified fast retransmit may make some of the lksctp bundling strategies superfluous for the first retransmission. The small share of spurious retransmissions when timeouts occur indicates that the lowered RTO_{min} is an effective means for recovery for such thin streams. For the second retransmissions, the share of spurious retransmissions is very large (more than 90%). As for unmodified TCP, the majority of second retransmissions is from timeouts. Even though most retransmissions are spurious, the share is not much larger than for unmodified lksctp. The relative share of fast retransmissions has increased compared to unmodified lksctp, while bundles have decreased somewhat. The high share of spurious retransmissions indicates that the high IAT makes it difficult to calculate a good RTO, resulting in retransmissions by timeout just too early. For the third retransmission group, the general pattern resembles closely that of unmodified lksctp. The overall share of spurious retransmissions is increased. For fast retransmissions in this test, none of the third retransmissions were valid. The share of fast retransmissions is low, however. We can see large improvements in the ACK latency for modified SCTP. Both average and maximum values show large reductions compared to unmodified lksctp.

The increase in spurious retransmissions seen when the modified SCTP is applied is not very large. The second and third retransmissions also occur seldom, which means that the count of packets containing redundant data is low. For such thin-stream scenarios, we think that the large improvement in retransmission latencies, reducing maximum latencies from more than 4000 ms to a little above 2000 ms for our modifications (see figure 5.26), justifies the additional spurious retransmissions that occur.

[12] In some scenarios spurious retransmissions may help to keep the congestion window open by providing extra feedback to the sender.

5.3.5 Internet tests

To see if our modifications could also improve the latencies observed at the transport layer in a realistic, real-world scenario over the Internet, we replayed game traffic from Anarchy Online (see table 2.1) between machines in Oslo and a machine located at the University of Massachusetts (Amherst, MA, USA). We ran 12-hour tests both from our university network and from three Norwegian ISPs (Get, NextGenTel and Telenor). We measured a minimum RTT of 121 ms and loss rates below 0.1%. Experiments were performed with all modifications. The modified lksctp was based on the 2.6.16 Linux kernel, while the unmodified lksctp in our tests was from the newer 2.6.22 Linux kernel[13]. FreeBSD 6.2 was also evaluated for this test scenario. As can be seen in figures 5.27 and 5.28, we observed varying loss rates and patterns that provided different conditions for SCTP, and the results show that the proposed modifications generally improved the transport-layer latency (and thus the QoE) when loss occurs and retransmission becomes necessary.

Figure 5.27 shows the results when we compare lksctp and SCTP in FreeBSD, with and without the early fast retransmit (EFR, see section 3.8), to our modified SCTP. To ensure that the network conditions were comparable, we had four machines, one for each setup, concurrently sending game traffic to a machine running unmodified lksctp at UMass. We used tcpdump on both sides and calculated the delivery delay as described in section 5.1.9. The delivery latency for the set of tests presented in this section is transport-layer latency. The transport-layer latency calculations which takes the in-order requirement into account was not implemented for our SCTP analysis software due to time limitations. We expect that the application-layer delivery latency follows the same patterns as we have seen in the TCP-experiments in section 5.2 since the basic mechanisms for retransmissions are the same for SCTP as for TCP.

Figure 5.27 shows a CDF of the arrival times, i.e., the amount of data (in number of bytes) that has arrived within a given latency (in milliseconds). Large deviations from the average occur only when retransmissions are necessary. In this test, we experienced a packet loss rate below 0.1% which means that the setups perform more or less equally up to a CDF of 0.999. This is also confirmed by the statistics shown in table 5.6 which show that all tests have similar average latencies. As shown in figure 2.3, higher loss-rates can be expected in a game-server setting, and even low loss rates can cause QoE-degrading latency events.

When loss is experienced, the differences are clear. For a small but relevant number of packets that are retransmitted by fast retransmit, lksctp achieves lower latencies than FreeBSD. FreeBSD with EFR follows unmodified FreeBSD closely for most situations. It has however clear benefits over both lksctp and unmodified FreeBSD for a relevant number of chunks that

[13] We used the unmodified 2.6.22 kernel as comparison since it was the newest kernel at the time the experiments were performed, and updates to SCTP had been implemented that could potentially improve the performance. The observed results did not encourage a port of our extensions.

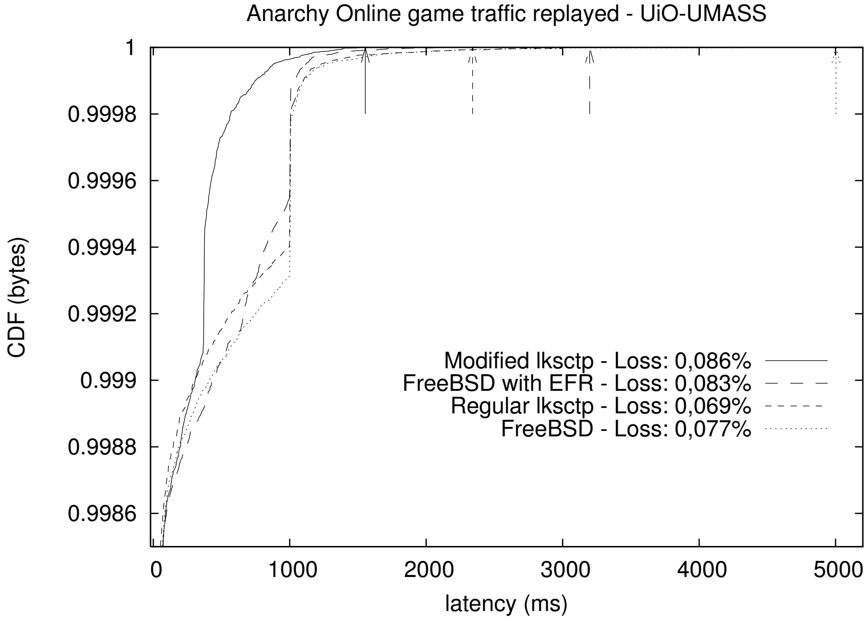


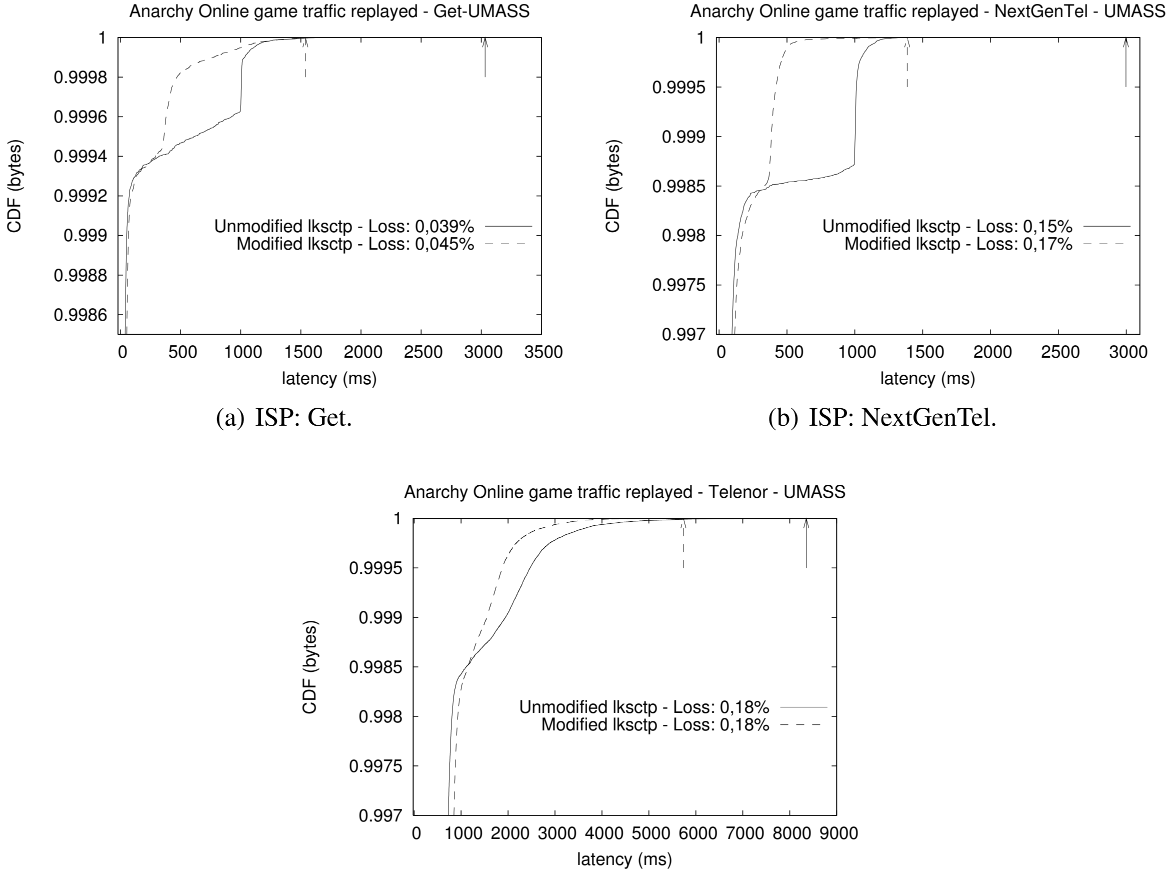
Figure 5.27: CDF of transport-layer delivery latency for replayed Anarchy Online game traffic.

	loss rate (%)	spurious retransmissions (%)	average latency (ms)	maximum latency (ms)
mod. lksctp	0.0855	6.708	302	1725
lksctp	0.0690	0.032	304	3521
FreeBSD	0.0765	0.006	303	5326
FreeBSD EFR	0.0831	0.038	304	2664

Table 5.6: Relative arrival time statistics for $\sim 2.65 \times 10^6$ packets.

are early-fast-retransmitted (in the CDF range 0.9992 to 0.9995). That these benefits do not have a larger effect on the CDF is most likely caused by the small number of packets that are concurrently in-flight in our scenario. That inhibits the re-opening of the congestion window when it has collapsed, which in turn prevents EFR from being triggered at all because the condition is that flight size must be smaller than the congestion window size.

Modified lksctp delivers a considerable number of chunks with shorter latency, and looking at the maximum latencies experienced (shown by the arrows in figure 5.27 and in table 5.6), we see large improvements. The latency improvement is mainly due to removal of the retransmission timer reset after reception of a partial SACK, which forces all other SCTP variations to wait one RTO_{min} before retransmitting lost chunks in idle phases of the sender application. Considering that the minimum RTT for the connection was 121 ms, this demonstrates that the



(a) ISP: Get.

(b) ISP: NextGenTel.

(c) ISP: Telenor.

Figure 5.28: CDF for transport-layer delivery latency for replayed Anarchy Online game traffic between UMass and three different commercial access networks in Oslo (Norway).

modifications can reduce the transport-layer latency of a relevant number of lost chunks by several RTTs.

As shown earlier (for example in figure 5.26(b)), the latency improvement comes at the cost of a slightly increased bandwidth requirement. Table 5.6 shows that the modifications increase the number of spurious retransmissions compared to all the other tested mechanisms. Nevertheless, for the interactive thin-stream applications of our scenario, both the increase in bandwidth and the collapse of the congestion window are negligible disadvantages compared to the latency reduction that can be achieved.

The tests above were performed at our university and may thus not represent the network conditions of a typical user. We validated the results in typical home user settings by running the Internet tests also from three typical access networks provided by Norwegian ISPs. As lksctp and SCTP in FreeBSD (with and without EFR) had similar performance, we compared only modified and unmodified lksctp. The results are shown in figure 5.28. We see the same trends. Our modifications reduce the transport-layer latency in case of loss, and as shown by the arrows in the plot, the devastating worst case delays are reduced on the order of seconds.

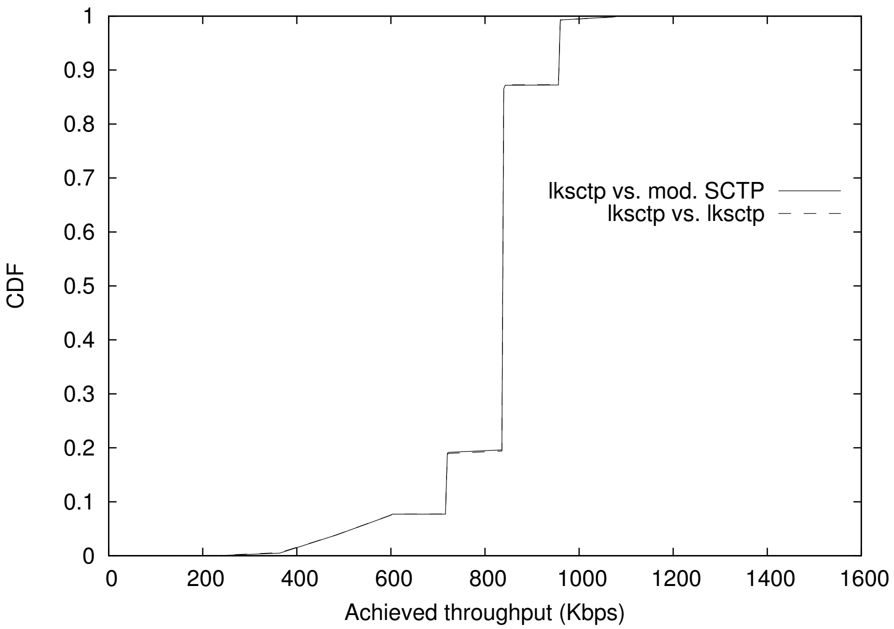


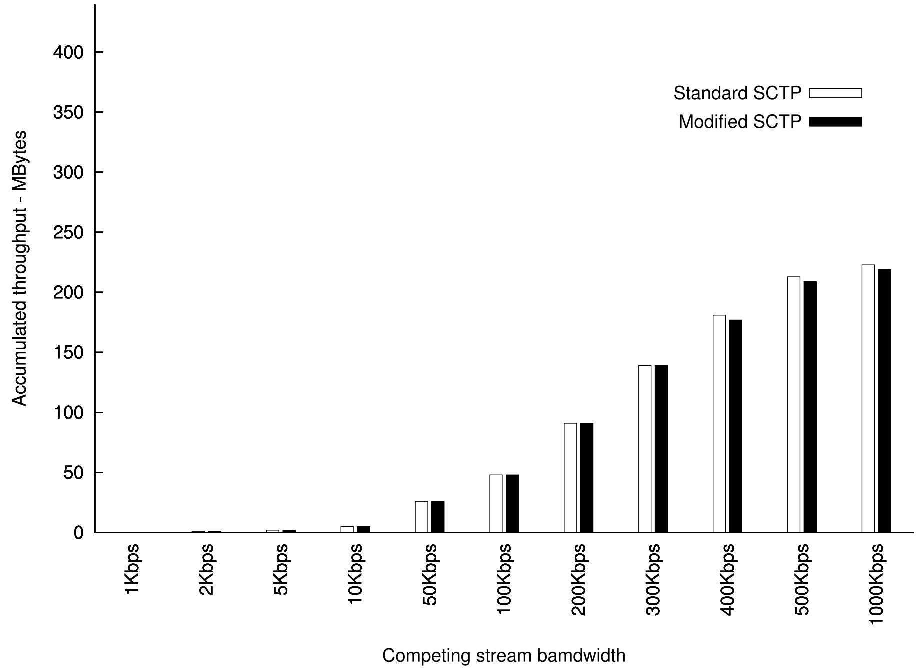
Figure 5.29: CDF of throughput on 100 ms intervals (connection RTT) for lksctp vs. lksctp and lksctp vs. modified SCTP.

5.3.6 Fairness

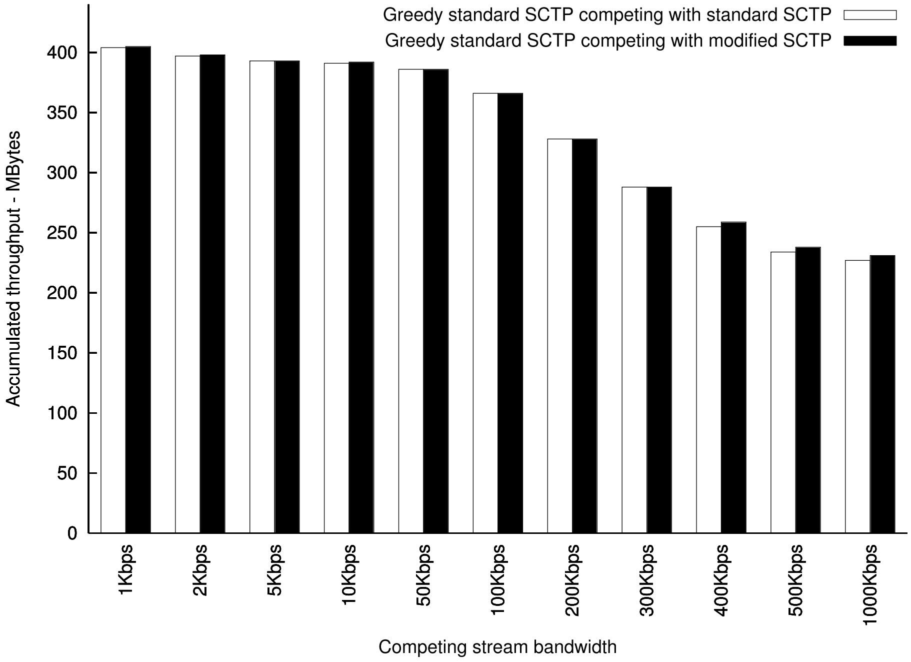
A major concern when modifying a transmission protocol like SCTP is whether the principle of fairness for congestion-controlled protocols is preserved. This is especially important in our case, in which more aggressive retransmission measures are implemented. To determine the degree to which the new mechanisms affect fairness, we set up a range of tests where regular SCTP (lksctp) streams competed with modified SCTP. For reference, we also tested two competing lksctp streams. We used the testbed shown in section 5.1.3, introduced a 50 ms delay in each direction and limited the bandwidth to 1 Mbps. The streams' achieved throughput was compared as a metric for fairness.

Figure 5.30(a) shows the aggregated throughput of the lksctp stream and the modified SCTP stream when trying to achieve different send rates in competition with a greedy lksctp stream. The figure shows no noticeable difference at the "thin-stream" rates. When bit rates increase, and the modifications are no longer active, the regular lksctp actually achieves a little higher throughput than the modified SCTP. This can be explained by small delays in the modified SCTP code that are introduced by the data structures for handling loss and packets in transit. In addition, there are tests to establish whether a stream is thin that are not present in regular lksctp.

In figure 5.30(b), the throughput of the greedy streams competing with modified and unmodified SCTP is shown. The graph shows also here that the throughput is nearly identical. As previously explained, the stream competing with the modified SCTP has slightly higher



(a) Increasing bandwidth stream throughput.



(b) Greedy stream throughput.

Figure 5.30: Comparison of throughput as an indication of fairness.

throughput in the 400, 500 and 1000Kbps experiments. Furthermore, measurements were performed to calculate the average throughput every two seconds to see the short term variations. An example of this is shown in figure 5.29 where only very small differences can be seen between the throughput of the stream that competes with regular lksctp and the stream that competes with the modified SCTP.

The tests indicate that fairness is preserved when a modified SCTP stream competes with an lksctp stream; actually, the stream competing with our modified lksctp achieves slightly higher aggregated throughput. When few packets are sent per RTT, few resources are consumed whether our modifications are in use or not. When the number of packets per RTT grows, the consumption of resources is almost identical. The reason is that our modifications are switched off when the number of packets in transit exceeds the threshold for thin streams.

5.3.7 Summary

In this section, we have presented the results of our experiments to determine the effects of our thin-stream modifications to SCTP. Experiments have been conducted in a laboratory where parameters like loss, RTT and IAT have been varied. As a part of the laboratory experiments, the statistics for which mechanism triggered each retransmission have been analysed. Traces from thin-stream applications have been replayed over the Internet to measure the effect of the modifications under more realistic conditions. The following general conclusions can be drawn from our results:

- The high RTO$_{min}$ used in SCTP results in very high latencies for thin-streams in general.
- We can improve the thin-stream latency by choosing a lower RTO$_{min}$ (for instance 200 ms, as is used for TCP in Linux).
- Delayed acknowledgements influence the RTO calculation when the RTO$_{min}$ is reduced, and results in too high RTO values for thin streams.
- Thin streams using SCTP suffer from high retransmission latencies because of the inability to trigger fast retransmissions.
- Our thin-stream modifications to SCTP significantly reduce the retransmission latencies for most thin-stream scenarios.
- The LT mechanism greatly reduces maximum latencies (caused by several consecutive retransmissions by timeout).
- Experiments where we compare the achieved throughput for competing SCTP streams (with and without the modifications) show that no disadvantage to fairness is detected.

Test	Description
ENet	Unmodified ENet gaming framework over UDP
modified ENet	ENet with linear timeouts
UDT	UDT middleware over UDP
TCP New Reno	Unmodified TCP New Reno
TCP BIC	Unmodified TCP BIC
modified TCP	TCP with the LT and mFR modifications
TCP with RDB	TCP with only the RDB modification
SCTP	Unmodified lksctp
modified SCTP	SCTP with modified RTO$_{min}$, LT and mFR

Table 5.7: Mechanisms that were tested and compared.

In summary, our modifications to SCTP improve the application-layer latency performance for thin streams over the original lksctp and FreeBSD implementations of SCTP, regardless of loss pattern and RTT. While the average latency is nearly unchanged (at least for very low loss rates), we are able to handle a large number of the cases that are caused by multiple packet losses and that cause severe application-layer delays for interactive applications. This is achieved through slightly more aggressive retransmission strategies when thin streams are detected by the system. A slightly increased number of spurious retransmissions pays for significantly improved thin-stream latency.

5.4 UDP and application layer approaches

Many application-layer frameworks are available that supply reliability and in-order delivery on top of UDP. In many cases, such frameworks model their mechanisms on the ideas from TCP, although UDP places no limits on the send rate. If the frameworks that supply such services are tailored for interactive applications, like games, it is in the interest of the user that the stream withdraws somewhat when heavy congestion is detected. We wanted to compare the gaming framework ENet and the advanced middleware UDT (described in section 3.5), as representative examples of such application layer approaches, to transport-layer protocols to determine their effect on thin-stream latency. After studying the source code for ENet, we found that exponential backoff is practised for retransmissions. We therefore also made a modified version of ENet that uses linear timeouts. UDT is designed for high-speed, wide area networks, and uses its own congestion control and reliability mechanisms. It probes the path for bandwidth, and redundantly uses what it identifies as “spare” bandwidth to improve throughput and latency.

For comparison, we tested several of the protocols and mechanisms described in chapter 3. Table 5.7 shows the frameworks and transport protocols that were tested and compared. TCP New Reno had no options turned on whereas BIC used SACK, FACK and DSACK. SCTP uses

RTT (ms)	50, 100, 200
Loss	0.1%, 0.5%, 2.5%
IAT (ms)	50, 100,200
Packet size (B)	100

Table 5.8: Parameters for experiments performed on UDP and application-layer reliability frameworks.

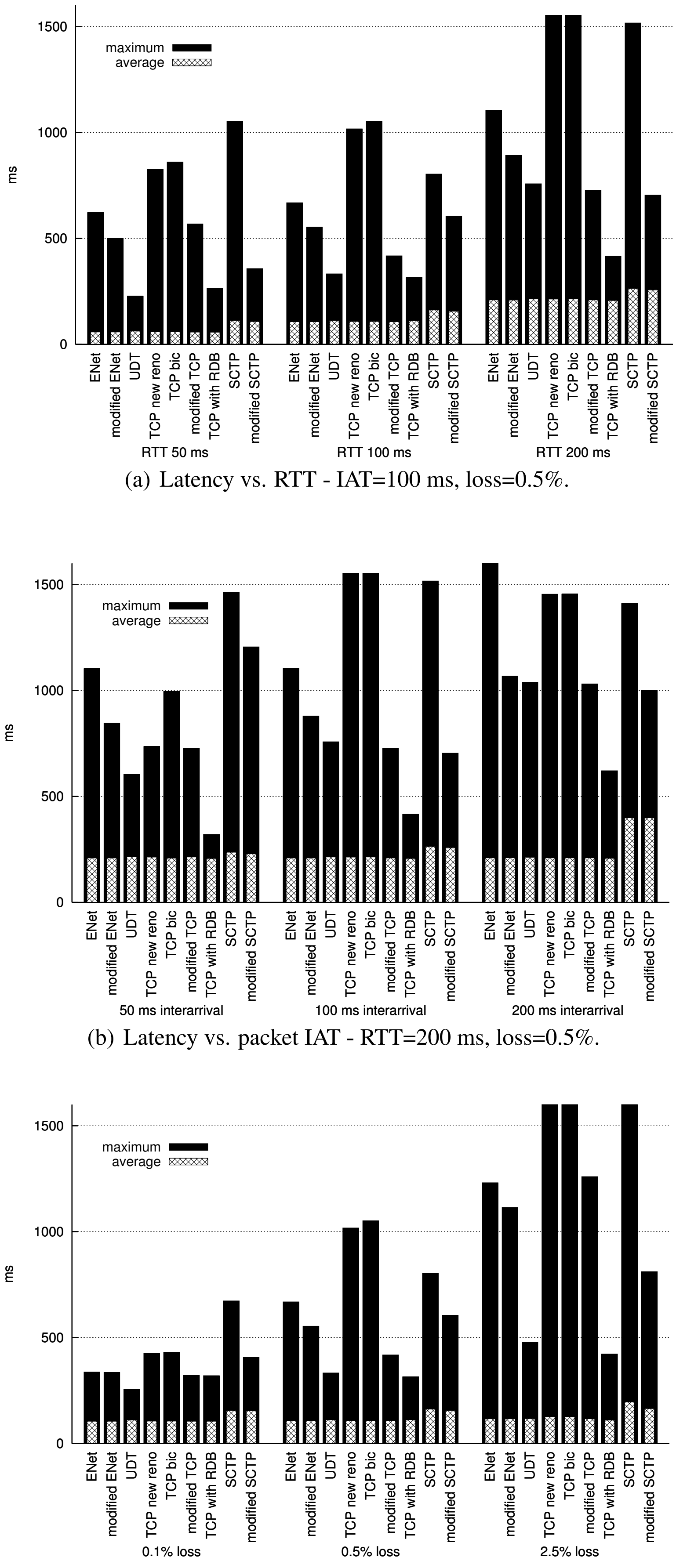
SACK by default. For this interactive scenario, we turned off Nagle's algorithm (explained in section 3.1.1) for both TCP and SCTP. Additionally, we tested TCP and SCTP with thin-stream modifications and the RDB algorithm.

Table 5.8 shows the parameters for this set of experiments. All combinations of the listed parameters were tested. Small packets were sent at a low rate in accordance with the game characteristics described in section 2.1. To emulate the network, we used *netem* to introduce delay and packet loss as described in section 5.1.2.

5.4.1 Latency comparison

One of the most important aspects of networking support for an interactive gaming scenario is the system's ability to deliver data in time. The measured results for the different mechanisms listed above are shown in figure 5.31[14]. The first observation is that the average results are very similar, except for SCTP, which generally has higher latencies. Thus, with respect to average latency, all the TCP and UDP based approaches seem usable. However, looking at the worst case scenario, which really hurts the user experience, we can see large differences. These maximum latencies depend on the number of retransmissions of the same packet. Since the loss is random, the maximum number of consecutive lost transmissions of a packet varies. The figures give nevertheless a representative picture of the performance of the different mechanisms. The plots in figures 5.31(a), 5.31(b) and 5.31(c) show the latency varying the RTT, packet IAT and loss, respectively. When the IAT is equal to (or higher than) the RTT, we see that retransmissions from timeouts and backoffs result in very large values for ENet, TCP New Reno, TCP BIC and SCTP. By changing the retransmission mechanisms as described when applicable, we can achieve large latency improvements. This comes at the earlier mentioned cost of a possible increase in spurious retransmissions. Another (orthogonal) way to improve the latency is to send multiple copies of a data element by bundling unacknowledged data in succeeding packets like in UDT (when using less than the estimated bandwidth) and TCP with RDB. The modifications increase aggressiveness in (re)transmissions, and may have an impact on fairness. We therefore next describe the results from experiments where we examine the bandwidth tradeoff from these changes.

[14] When the loss rate is high (2.5%) and the RTT is high (200ms), the standard TCP variants and SCTP have maximum values well above 2000 ms, although the scale of the figure is limited at 1500.



(a) Latency vs. RTT - IAT=100 ms, loss=0.5%.

(b) Latency vs. packet IAT - RTT=200 ms, loss=0.5%.

(c) Latency vs. loss rate - RTT=100 ms, IAT=100 ms.

Figure 5.31: ACK latency.

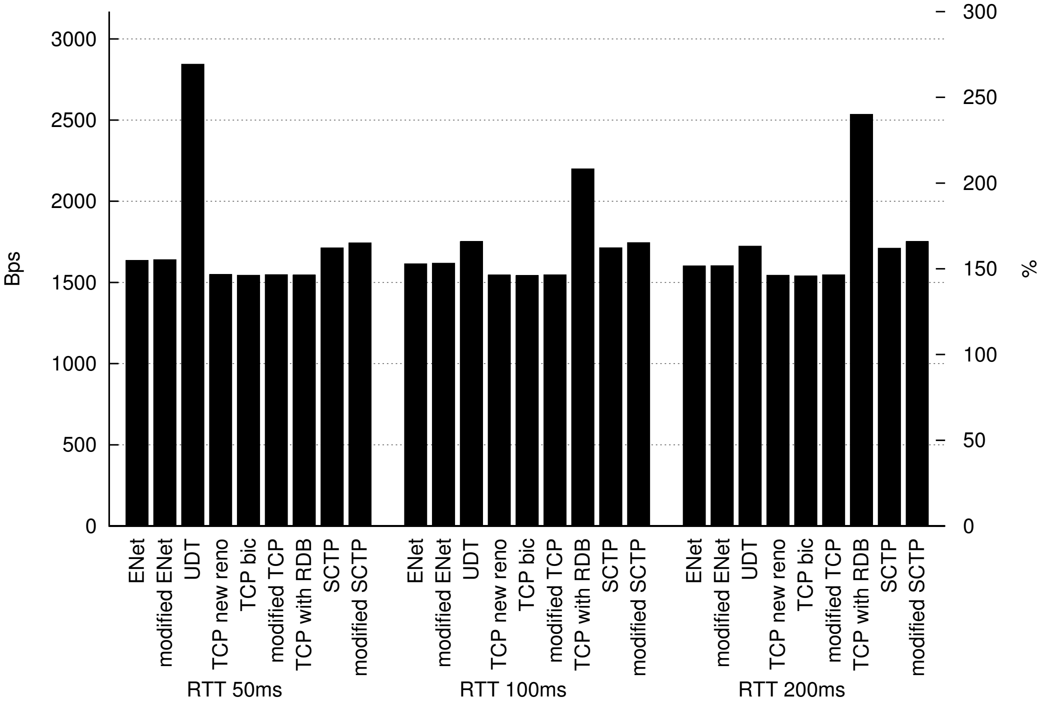


Figure 5.32: Measured bandwidth - IAT=100 ms, loss=0.5%.

5.4.2 Bandwidth consumption

Adding support for reliability comes at the price of retransmitting lost packets, and trying to reduce the retransmission latency increases the possibility of spurious retransmissions, which increases the bandwidth requirement further. To quantify the tradeoff, we have measured the number of bytes per second (Bps) and the number of packets per second (pps). Figure 5.32 shows the required bandwidth corresponding to the achieved latencies in figure 5.31(a) where the packet IAT is 100 ms and the loss rate is 0.5% in both directions. On the right y-axis, the figure shows the relative bandwidth compared to the IP payload of pure UDP packets. With respect to the number of bytes sent, the traditional TCP variants are best regardless of loss, packet interarrival time and RTT. Compared to the user space libraries, TCP and SCTP do not add an additional header, and therefore consume marginally less bandwidth. However, the data and loss rates are low, so the increase is negligible. The most resource consuming approaches are UDT and TCP with RDB. The reasons are that UDT always tries to use all the estimated bandwidth, and RDB bundles previous packets as long as there is unacknowledged data in the queue and the size limit for the new packet is not reached. Thus, when the RTT is low, UDT re-sends a packet multiple times to fill the pipe, and the overhead for TCP with RDB increases with the amount of unacknowledged data. In terms of the number of bytes, TCP with RDB is more expensive with higher packet rates and longer RTTs. In contrast to UDT, RDB would remain active even if no "free" bandwidth was detected over the network path. In such cases

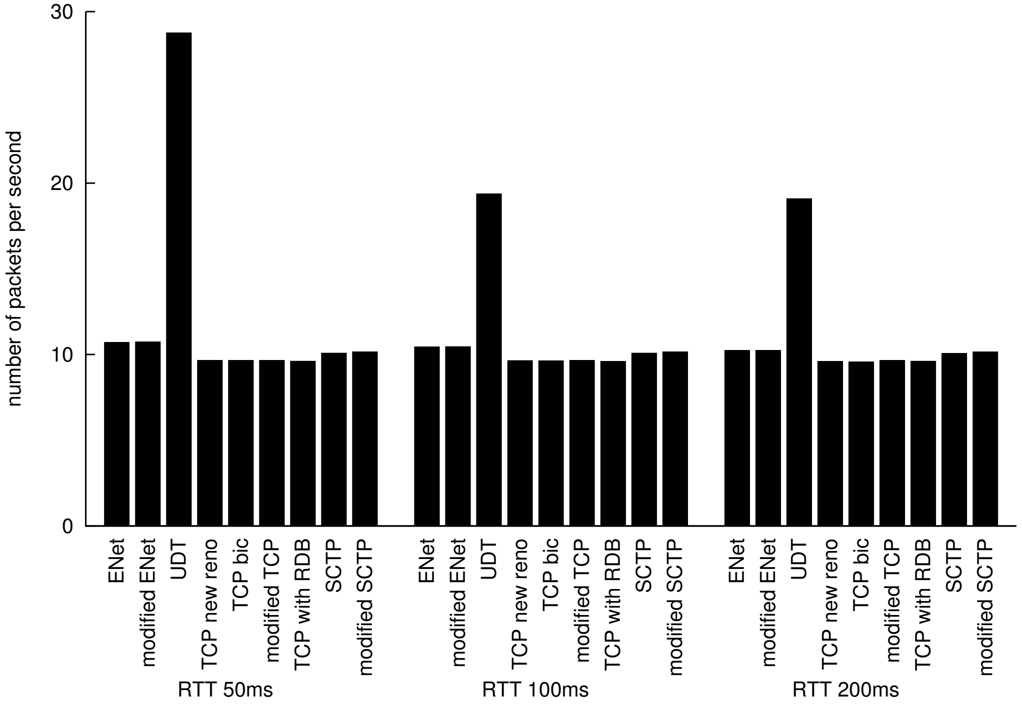


Figure 5.33: Number of packets sent - IAT=100 ms, loss=0.5%.

UDT would no longer provide lower latency for the application.

Another way of comparing the overhead is to look at the number of packets sent (see figure 5.33). Many consider this more relevant because of the fixed time between packets on the transmission medium. For example, the minimum size of an Ethernet frame is 64 bytes corresponding to the Ethernet slot time of 512 bits used for carrier sensing and collision detection at 10 and 100 Mbps. For Gigabit Ethernet, the slot is increased from 512 to 4096 bit. Thus, it can be said that space may be wasted if the packets are not filled – at least to the slot size (see section 4.1.5).

In our plots, the application sends approximately 10 pps (actually marginally less on average due to the timing granularity in user space). Traditional TCP follows this rate and since only a few packets are lost, the measured packet rate is approximately 9.6 pps. ENet and SCTP (as well as the modified versions) both send a few more packets. As UDT always try to fill the estimated pipe, the packet rate is large, e.g., for an RTT of 50 ms, UDT sends about 29 packets per second. Finally, TCP with RDB sends slightly fewer packets compared to standard TCP since no retransmission was triggered using RDB.

5.5 Summary

In this chapter, we have presented results from a large number of experiments performed both in a laboratory environment and on the Internet. The experiments have compared traditional versions of TCP, SCTP as well as UDP with application layer approaches to modified versions designed to reduce latency for thin streams. With respect to data delivery latency, the proposed enhancements generally reduce latency and especially the large worst-case delays. This is achieved at the price of a slightly increased bandwidth. According to our fairness-experiments, the modifications will hardly affect per-stream fairness. An exception is RDB, which can grab some more resources than unmodified TCP for some special cases. In such cases of relatively low IAT and small packet sizes, the applications may suffer large delays due to the inadequacies of the retransmission mechanisms of TCP, and are significantly helped by the redundant transmissions of RDB. We therefore think that our enhancements are viable, and look at how their improvements in data delivery latency influence the users' QoE next.

Chapter 6

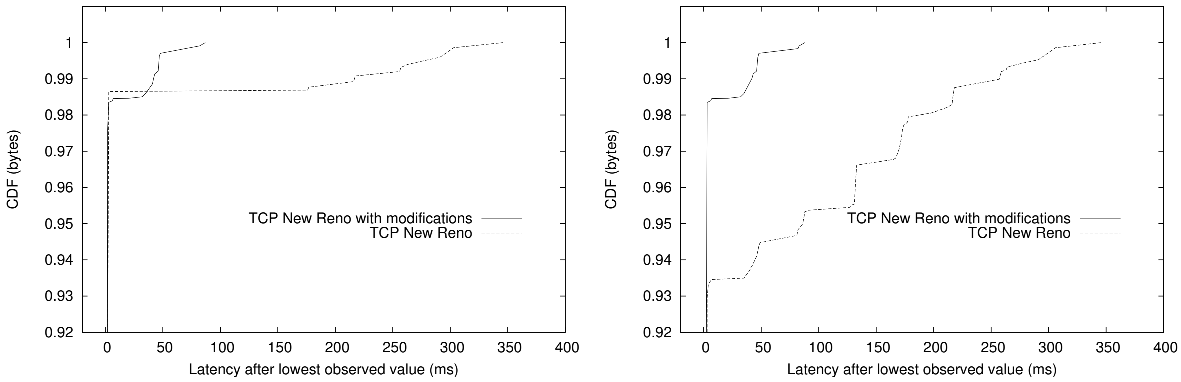
Experienced effect for the users

After performing the extensive measurements of our TCP modifications presented in section 5.2, a natural way to substantiate the observed measurements was to perform subjective user evaluations. We have therefore carried out user surveys evaluating the QoE for some thin-stream interactive applications. We have also performed a laboratory experiment where we determined the chance of hitting an opponent in an FPS game (BZFlag [4]) with and without our modifications. Finally, we have created an interactive demonstration where the effect of each of our TCP modifications can be tested in a thin-stream game setting.

6.1 Skype user tests

IP telephony over the Internet is used by a steadily increasing number of people. We wanted to investigate whether our modifications could improve the perceived quality of such a VoIP session, and chose Skype as the test application. Skype is a popular VoIP program that defaults to UDP for transport, but falls back to TCP if UDP is blocked for some reason. As shown in table 2.1, it generates small packets and has a relatively high packet IAT, which makes it a typical thin stream application. The TCP modifications for thin streams should be able to help reduce latency upon packet loss. Due to the relatively low IAT (e.g., compared to Anarchy Online in table 2.1), the triggering of fast retransmissions by standard mechanisms is possible.

Considering speech in VoIP conferences, differences between each conversation can make it difficult to evaluate one session when compared to another. To have directly comparable data, and to be able to reproduce the results, we chose to use sound clips which we played across the Skype session. We experimented with several different sound clips, both with respect to the numerical results gathered from packet traces, and to feedback from users in preliminary tests. A total of three different sound clips were ultimately chosen for this test. Each clip was played two times, once using our TCP modifications and once using unmodified TCP. The sound clip was sent across the Skype connection and the resulting output was recorded at the receiver.



(a) Transport layer latency.

(b) Application layer latency.

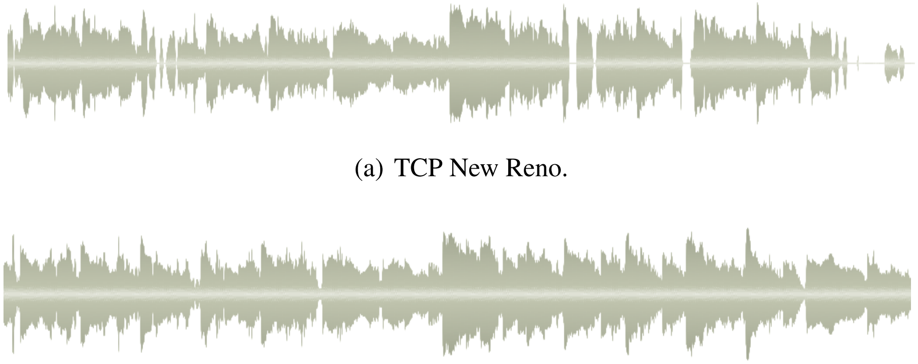
Figure 6.1: Skype latency - loss=2%, RTT=130 ms.

When performing the user evaluation experiment, we played the first minute of each clip. The length was chosen to make it possible to remember the perceived quality for comparison. We present the results from one test where a speech news podcast was used as source, and two tests where music was used. The podcast was chosen because speech is what Skype is designed for, thus the codec should be tuned for it. The music was chosen because it is easier to notice audio artifacts when there is a steady rhythm to relate to. A complicating factor is that Skype encoding sometimes distorts the sound when using TCP, even under perfect network conditions (no loss or delay). All the recordings would, however, be exposed to these irregularities, so the resulting recordings should be directly comparable. Unfortunately, it was not possible to investigate the cause of the distortion further since Skype is a proprietary application.

As a reference to the improvements that we observed in statistical analysis, we included CDFs of delivery latency for one of the clips used in the experiment. The configuration of the environment used for the test was as described in section 5.1.2. Figure 6.1 shows both application and transport layer delivery latency. The statistics shown are very similar to what we have seen in other tests in section 5.2 where the gain at the transport layer greatly improves the results at the application layer.

When taking part in a phone conversation, one of the most important aspects is the sound quality. Distortion and similar artifacts degrades the user experience, making it more difficult to understand the speaker. We therefore made recordings of Skype conversations played over links with loss and delay, and had a group of people evaluate the perceived quality.

To get an impression of how distortions affect the sound, audio spectrums for one of the test clips used are shown in figure 6.2. The clip was played over a Skype connection with 2% loss and an RTT of 130 ms. One playback was performed using TCP with all our modifications active, and one using unmodified TCP New Reno. Where TCP New Reno is used (figure 6.2(a)), we can see several gaps in the audio waves that distort the sound experienced by the user. In figure 6.2(b), the same clip is played over the same link using our TCP modifications. We



(a) TCP New Reno.

(b) TCP with LT, mFR and RDB.

Figure 6.2: A 20 second "audio spectrum" of a representative Skype session played over a link with 2% loss and an RTT of 130 ms.

can observe distortions for this clip also, but at a much lower frequency and for shorter time intervals.

As a reference test, we played the same version of one sound clip twice. This was done to ensure that a "memory effect" does not influence the answers overly much (for instance that, after listening to two clips, the listener would prefer the first clip because he/she had "forgotten" the faults observed in that clip.)

In total, we collected 88 votes, and the results are shown in figure 6.3. The recordings made with the modifications were clearly preferred by the users. We were told that the differences in "clip 1", which was the podcast, were small but still noticeable. With clip 3, which was one of the songs, the users commented that the version without the modifications was distinctly suffering in quality compared to the clip where modified TCP was used. The test subjects complained about delays, noise, gaps, and others artifacts, and said that it was easy to hear the difference.

In the reference test ("Clip 2" in figure 6.3), the majority of the test subjects answered that they considered the quality as equal. Of the users that decided on one of the versions of this clip, most of them chose the one that was played first. This is caused by the "memory effect" (discussed above); the listener may have "forgotten" the errors of the first clip. For "Clip 1", the modified version was the second clip to be played. The "memory effect" may here have diminished the results for the modified version of TCP. Even so, a relevant majority of the test subjects preferred the modified version. For "Clip 3" (figure 6.3), the order was the opposite (modified TCP first). We can assume that some of the people who chose the modified TCP version were fooled by the "memory effect". However, the majority of subjects who chose the modified TCP version is so large (95,4%) that we deem the numbers reliable.

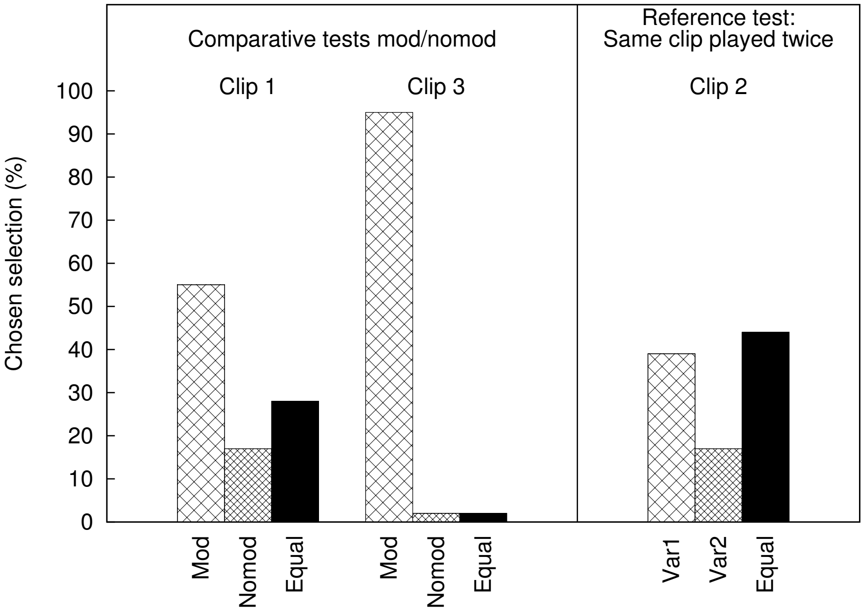


Figure 6.3: Preferred sound clips from Skype user tests.

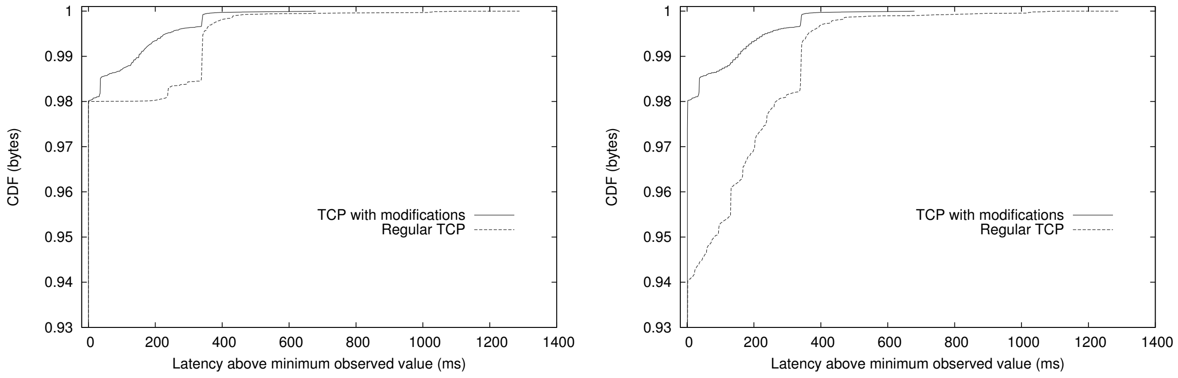
6.2 Secure shell user test

The second application that was chosen for user evaluation was secure shell (SSH). As shown in chapter 2, a text-based SSH session represents a typical thin stream. This protocol is widely used as a remote tunnel for example to administrate servers or remotely edit configuration files. While working in an SSH session, loss can lead to delays that make the editing of text more difficult (e.g., lag between pressing keys and the subsequent screen update).

The system used for this experiment was configured as described in section 5.1.2. We have included here the CDFs of a SSH session played over the test network so that the users' evaluations can be evaluated in view of the statistics. The resulting CDFs of transport and application latency are shown in figure 6.4. Here also, the statistics are very similar to the analysis presented in section 5.2.

The experience of using a remote text terminal can be severely diminished by network loss. The screen may not be updated with the character that was typed, and it may be difficult to edit the document in a controlled manner. After analysing the SSH latency, we wanted to test if the improvements in application layer latency could be noticed by the user.

The test network was configured in the same way as for the Skype test. The users opened a command window on the sender computer and initiated a text-based SSH connection to the receiver. Each user then opened a text editor (like "vi" or "emacs") and typed a few sentences. The users were encouraged to try to keep their eyes on the screen while typing in order to observe any irregularities that might occur while typing. In order to make the test applicable



(a) Transport layer latency.
(b) Application layer latency.

Figure 6.4: SSH latency - loss=2%, RTT=130 ms.

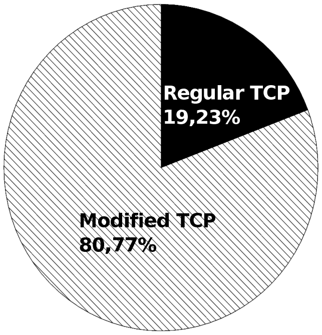


Figure 6.5: SSH user test: Preferred connection.

to the users that prefer watching the keyboard while writing, a second test was devised. This test consisted of repeatedly hitting the same key, while watching the screen. After the typing, the user was to close the editor and log out of the SSH-session. We then made a change in the network configuration (turned on or off the modifications), and the user repeated the procedure. Afterwards, the test subjects had to decide which of the two sessions they considered to have the best performance. In order to avoid the "memory effect", half of the test subjects took the modified TCP test first, the other half took the regular TCP test first. A total of 26 people participated in this test. All of the participants were familiar with text editing over SSH, and used a text editor that they were comfortable with.

Figure 6.5 shows how many chose each type of session. The black area represents the percentage of the test subjects who preferred the regular TCP session (19%), while the hachured area represents the ones who preferred the session that used the modified TCP (81%). A frequently occurring comment from the users was that they preferred the connection using the modified TCP because the disturbances that occurred seemed less bursty. That is, if they had written a group of letters, they showed up without a large delay. Another recurring comment was that deleting became more difficult in the sessions using regular TCP because it was easy to delete too many letters at a time. The large share of users that preferred the session using the modified TCP strongly suggests that the improvement shown in figure 6.4 can also be experienced at the user level.

The number of participants for the user test may, statistically, be too small to draw absolute conclusions based on them. Seen in correlation with the measured data, however, we feel that this is a strong indication of the impact of the TCP modifications on the user experience for the tested thin-stream application.

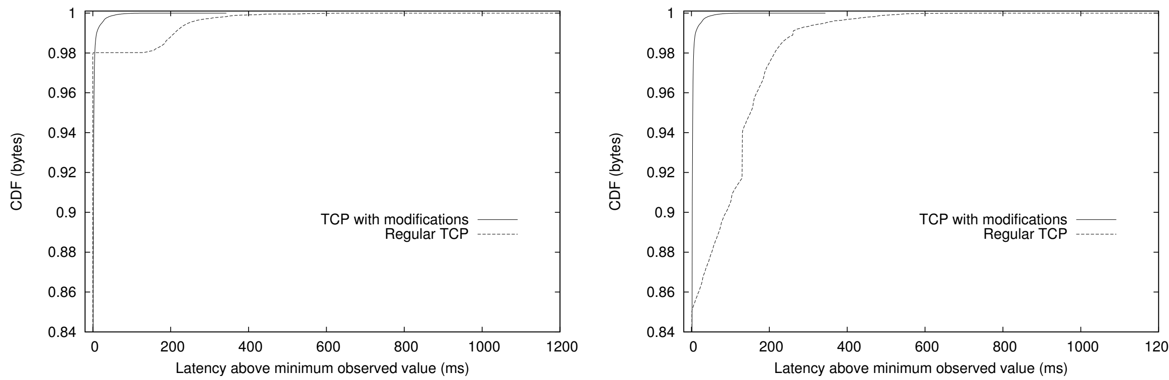
6.3 BZFlag hit probability evaluation

The TCP modifications are most effective for interactive thin-stream applications like online games. As a way to evaluate the effect of the mechanisms, we wanted to benchmark the position updates in a real game environment. BZFlag [4] is an open source, FPS game where players challenge each other using tanks on a virtual battleground. As shown in table 2.1, it generates thin streams with an average packet IAT of 24 ms and an average payload size of 30 bytes. Thus, it is a game well suited to demonstrate the benefits of our thin stream TCP-modifications.

To collect the data needed to generate the results presented here, we constructed a network consisting of five machines. Three acted as clients running computer controlled opponents (bots), one ran the server application, while the fifth machine acted as a network emulator between the server and the clients. We needed to impose loss and delay on the links between the server and the clients so that we could evaluate the effects of the modifications. Thus, the server was placed behind a network emulator as described in section 5.1.2. After performing several measurements from different Norwegian ISP's to machines in the US and various European countries, we chose an average loss rate of 2 % and a round trip time (RTT) of 130 ms, as representative emulated network values. The three clients ran 26 bots altogether, which represents a typical high-paced BZFlag-multiplayer scenario. To get a sufficiently large number of samples, six one hour long tests were performed (three with the modifications enabled and three without).

From the packet traces, we measured the delivery latency for the connections in the experiment (shown in figure 6.6). We include the statistics from both transport layer and application layer as a reference in addition to the measurements that we present on shot accuracy.

The reduced application layer latency also affected the user experience. In initial experiments, users were asked to evaluate the accuracy of their aiming. It soon became clear, however, that it was difficult to determine whether a miss was caused by network conditions or bad aiming. We therefore devised a system for measuring the impact of latency from logs of perceived and actual positions at the time a shot was fired. To see how the latency influenced the perceived player positions, we collected the actual and perceived position of the other players



(a) Transport layer latency.

(b) Application layer latency.

Figure 6.6: BZFlag delivery latency - loss=2%, RTT=130 ms.

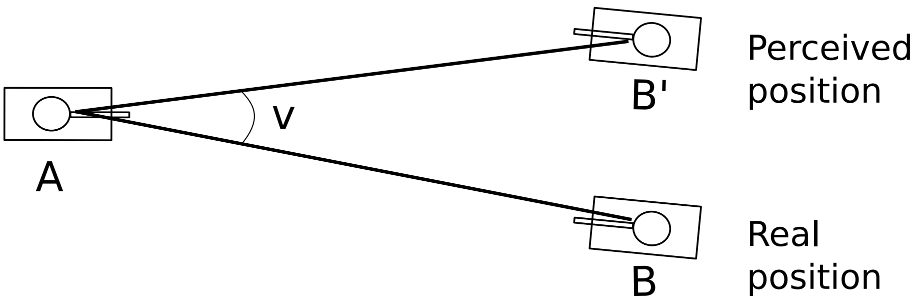


Figure 6.7: Difference angle calculation.

each time a chosen tank (reference tank) fired a shot. We then calculated the difference (angle) between the two positions as viewed from the reference tank. Figure 6.7 shows how the two vectors representing perceived and actual position were found. The angle v was then calculated using the law of cosines. Position A represents the reference tank, B represents the perceived position of an opponent at the time of the shot, while *B'* represents the actual position of the opponent. Figure 6.8 shows that the angle of difference between the line from the shooter to the perceived position of the target (AB' in figure 6.7) and the line to the actual position (AB in figure 6.7) is smaller when the modifications were applied. On average, the angle between AB' and AB was reduced by 0.7 degrees when the modifications were enabled (from 3.5 to 2.7 degrees). The dimensions of the game field (the virtual battleground) were 200 by 200 world units (wu, BZFlag's internal distance metric). In BZFlag, each tank is 2.8 wu wide and 6 wu long. Provided that the player aimed at the centre of the opponent's tank (a "perfect shot") based on the perceived position, the angle of difference (v) may be so substantial that a would-be hit actually evaluates as a miss. Figure 6.9 shows how we calculate the wu deviation caused by the angle v. Using the formula $x = n \times \tan v$, we can extrapolate the deviation in wu when the distance n to the target increases. Here, n is the distance between the player and the opponent, and x is the deviation from the actual position of the observed tank. A "perfect shot" would

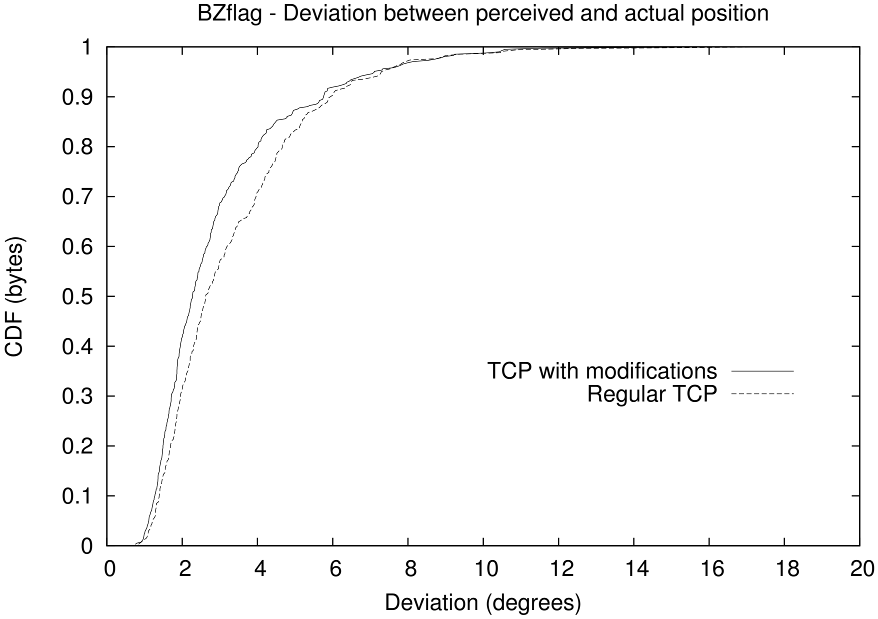


Figure 6.8: CDF of difference.

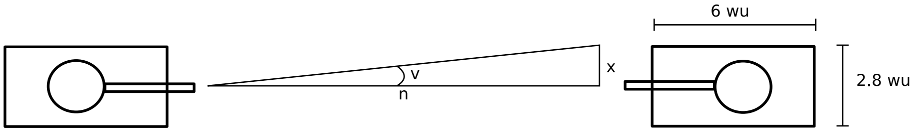
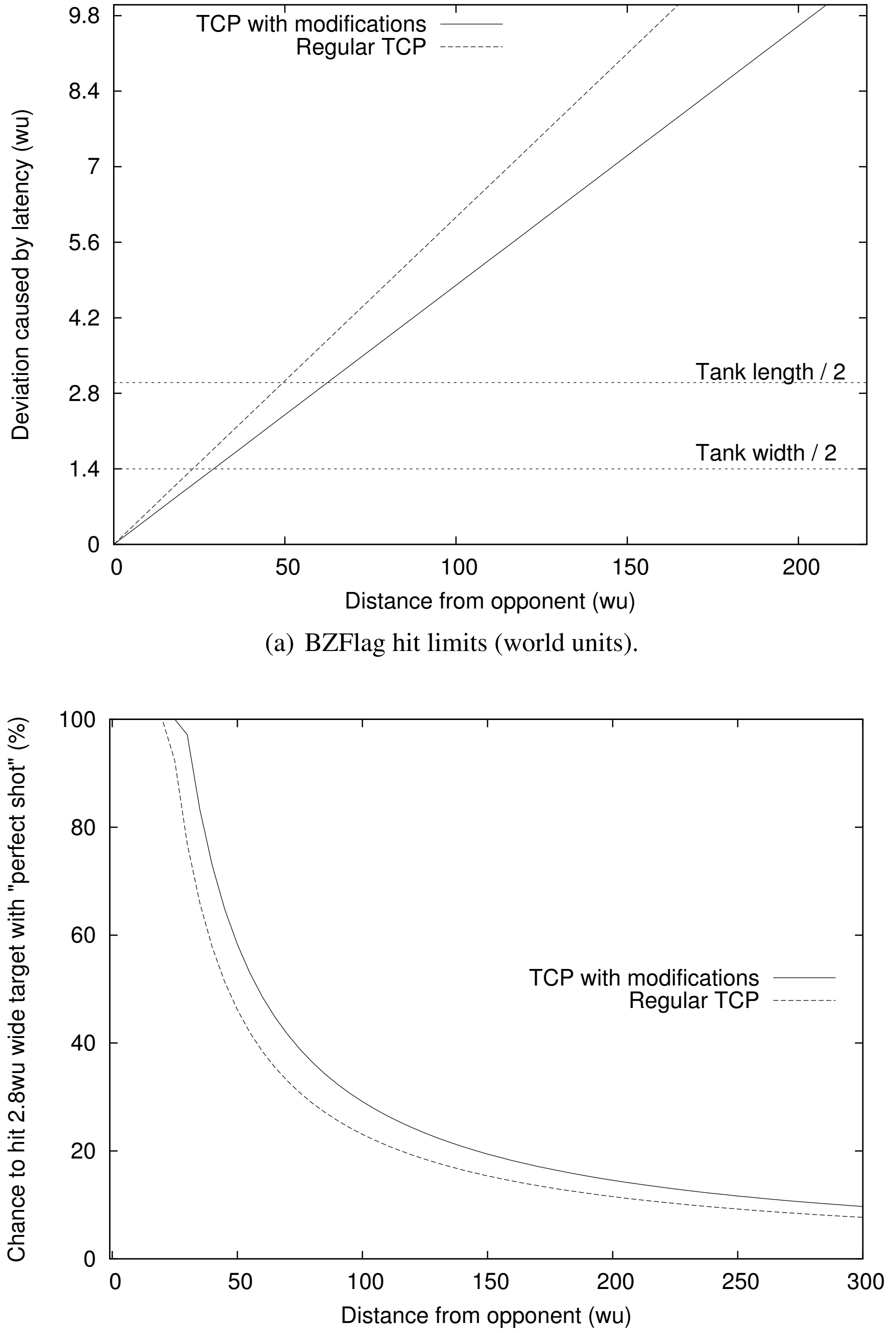


Figure 6.9: Hit limit calculation.

have a 100 % chance of hitting enemies when the distance to the target is less than 30 wu using the modifications. Using regular TCP, the distance to guarantee a hit would have to be reduced to 23 wu. In practise, this means that the modifications increase the chances of hitting your opponent due to a smaller deviation between perceived and actual position. The 7 wu improvement is, for reference, equal to the width of 2.5 tanks in the game, a deviation that is large when trying to hit your target. We also investigated how the difference between perceived and actual position affects the chance of the shot hitting when we varied the distance between the tanks. Figure 6.10(a) shows how large the radius of the hit box has to be to guarantee that the "perfect shot" is a hit given a specific distance to the opponent. The dotted lines at 1.4 wu and 3 wu represent the tank hit box when seen from the front and from the side. The graph shows how the effect of the modifications increases with the distance to the opponents you are trying to hit.

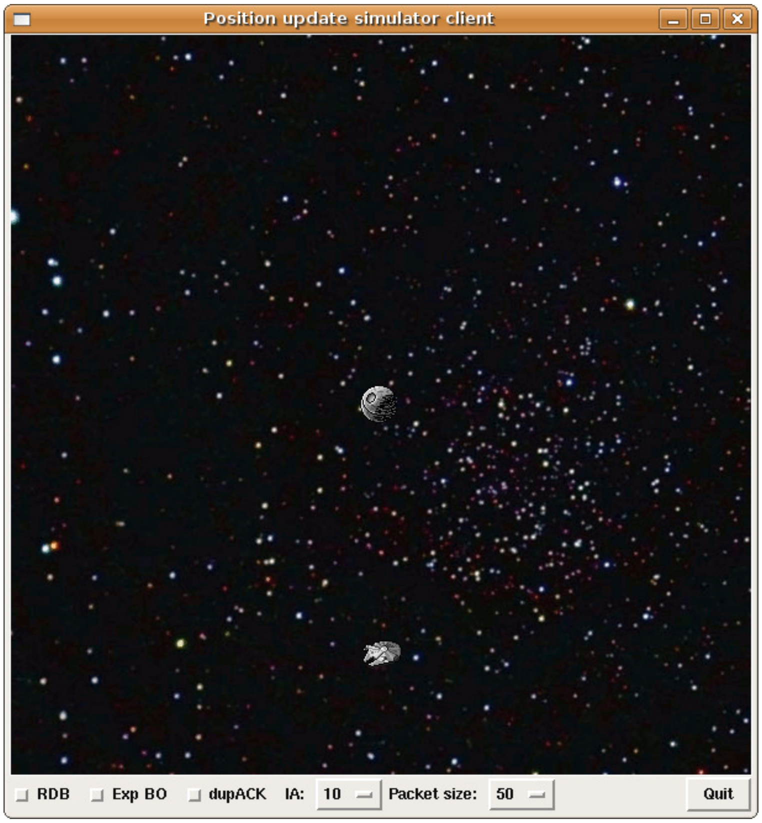
Figure 6.10(b) shows the chance of hitting your opponent with a "perfect shot" at different distances using regular TCP and the modifications. If you, as an example, fire at an opponent at a distance of 50 wu, there is 12 % greater chance of a hit using the modifications than without.



(a) BZFlag hit limits (world units).

(b) BZFlag hit limits (percent).

Figure 6.10: BZFlag hit limits for “perfect shot”.



□ RDB □ Exp BO □ dupACK IA: 10 Packet size: 50

Figure 6.11: Screen shot of the interactive, thin-stream, game demonstration

6.4 An interactive demonstration

In order to demonstrate the effects of TCP retransmission mechanisms on a thin stream (like position updates in a game), we have created a proof-of-concept application where the modifications can be dynamically turned on and off.

The server calculates the positions for an object moving in a circle, i.e., the Millennium Falcon circling the Death Star, in order to have a predictable trajectory. When a client connects, the server sends position updates with regular intervals. A network emulator positioned between the server and the client creates loss and delay[1]. The client, once connected, updates a graphical representation of the object each time a new position update is received. From the client user interface, several parameters can be changed in order to observe the effect on how position updates are received.

Interarrival time and packet size can be changed, and each of the thin-stream mechanisms described above can be turned on and off. When a network emulator is used to generate loss and delay, the RTT and loss rate also can be varied in order to experiment with different effects of our modifications.

[1] When this demonstration was performed at external locations, the network loss and delay were often sufficient to provide a good impression of the effect of our mechanisms. In such cases, the network emulator was not needed.

6.5 Summary

In this chapter we have described two user surveys that were performed as evaluations of the experienced effect of our TCP modifications. The subjective evaluation of the quality of Skype audio transfer and SSH text sessions both strengthened the assumption that there is a significant improvement to be observed at the application layer. We also calculated the difference in shot accuracy for a FPS game, and showed that the chance to hit your opponent increases when using our TCP modifications. Finally, we presented a demonstrator based on the concept of game position updates. In all experiments, the conclusion is clear: the modifications greatly improve the user's perceived quality in the thin-stream scenario.

Chapter 7

Conclusions

In this thesis, our work on reducing thin-stream latency for reliable network transport has been presented. We will now summarise the work presented herein, provide a critical review of the presented topics and outline possible directions of future work.

7.1 Summary

Distributed, interactive applications often produce what we call thin streams: data streams with high packet IATs and small packet sizes. Such streams, when transported over reliable protocols, result in high retransmission latencies when loss occurs. We analysed a selection of time-dependent applications to reveal their thin-stream properties, we have performed a thorough analysis of the latency behaviour of thin-stream gaming scenarios and we have evaluated the application-layer latency over UDP for the thin-stream scenario. Based on the observations from these analyses, we have modified TCP and SCTP to improve the thin-stream latency when using these protocols.

Our thin-stream modifications were implemented in the Linux kernel and subjected to extensive experiments. Using both laboratory experiments and Internet tests, the effects of the modifications were analysed. A set of user surveys was performed to determine the effect of our modifications on user perception. Furthermore, we performed an analysis of the actual effect of our modifications on the probability to hit a target in an FPS game. Finally, the effects of the implemented modifications were illustrated through an interactive demonstration.

7.2 Contributions

Here, we summarise the most important contributions derived from this work and relate our results to the hypotheses presented in section 1.3.

Investigation of thin-stream properties: The investigation of *Hypothesis 1* demanded that we performed in-depth analysis of data patterns from a wide range of interactive applications. The analysis we performed confirmed that a wide range of the time-dependent applications showed thin-stream properties. The variation in packet IAT and sizes for the thin streams was analysed with respect to their effects on delivery latency. Typical packet sizes ranged from 100 to 400 bytes, and IATs ranged from 30-600 ms.

Thin-stream latency analysis: To address *Hypothesis 2*, extensive analyses of latency for typical thin-stream scenarios were performed. From these analyses, we were able to determine that thin streams suffer from extreme latencies. In a trace from the massively multiplayer online game (MMOG) Anarchy Online [44], for example, we found acknowledgement latencies of up to 67 seconds. By studying the analysed traces in detail, we were able to determine the main reasons for the increased latencies. Furthermore, we performed experiments to determine whether some reliable transport protocol variations reduce delivery latency for thin streams. We identified TCP New Reno as the best alternative for reliable thin-stream transport, but we concluded also that none of the tested variations provides satisfactory latencies for thin streams.

Adaptation of retransmission mechanisms to reduce latency for thin streams: We implemented modifications to the existing retransmission mechanisms (in the Linux kernel) that reduce latency for thin streams. The mechanisms include timer reset corrections and a new RTO_{min} value for SCTP, fast retransmission after one dupACK as well as linear timeouts for both TCP and SCTP and a bundling mechanism for TCP. The mechanisms are dynamically switched on and off so that traditional mechanisms are used for greedy streams. The modifications were evaluated thoroughly through a series of experiments. In answer to the questions posed by *Hypothesis 3*, we found that the modifications are able to provide lowered delivery latency for thin streams.

Implementation of a bundling mechanism that takes advantage of small packet sizes in thin streams to reduce latency: We implemented a new mechanism that takes advantage of the fact that many thin streams have very small packet sizes. The bundling mechanism sends unacknowledged data segments with new packets, so as to avoid retransmissions. In many cases (like for Gigabit Ethernet), the minimum frame size is much larger than the packet sizes produced by thin-stream applications, making bundling possible with very little actual overhead. In answer to *Hypothesis 4*, the mechanism was evaluated through extensive experiments, and was found to significantly reduce delivery latency for thin streams.

Evaluation of transport protocols and our thin-stream modifications: We evaluated the described approaches for TCP, SCTP and UDP with application-layer reliability. All our modifications were designed to be transparent to the receiver in answer to *Hypothesis 5*; any unmodified (standards compliant) receiver can receive the benefit of a modified sender. Our findings show that we are able to reduce delivery latency for all the thin-stream scenarios we evaluated, especially the worst-case latencies that ruin the user experience are significantly reduced. In addition to the experiments performed to measure the latency of thin streams when using reliable transport, we performed surveys where users evaluated the effect of our mechanisms. All our results show that latency can be reduced significantly for thin-stream interactive applications by applying our mechanisms.

Evaluation of the impact of our modifications on per-stream fairness: As the implemented modifications apply more aggressive retransmission strategies when thin streams are detected, we evaluated also the effect of our modifications on competing streams (fairness). This evaluation showed that the modifications to the retransmission mechanisms do not affect fairness because the thin stream's congestion window stays below the minimum congestion window size. The bundling mechanism leads to increased packet sizes in certain scenarios, and therefore needs more resources. The number of sent packets, though, is not much higher since the bundling mechanism does not trigger extra transmissions.

The subject matter of this thesis resulted in five publications in peer-reviewed journals and conferences [54, 86, 87, 38, 85]. Additionally, the interactive demonstration was exhibited at NOSSDAV 2008 [89] and the thin-stream mechanisms and the Linux implementation were presented at the Linux Kongress 2008 in Hamburg [88].

The commonly used retransmission techniques, derived from the TCP specifications, result in high retransmission latencies for thin streams. Based on the findings from this thesis, we conclude that thin-streams should be handled separately from greedy streams when using reliable transport. We have shown that the high latencies can be significantly reduced by applying separate handling of thin- and greedy streams.

7.3 Critical assessment of the results

In the following, we review the hypotheses postulated in section 1.3 and evaluate each claim in view of the experiences gained from the work in this thesis.

- **Hypothesis 1:** *Thin streams are very often a product of time-dependent or interactive applications.*

During the course of our work on distributed, interactive applications, we found that such time-dependent applications very often show thin-stream properties (some of which are presented in table 2.1). When an application does not fill up the send buffers, it is usually because the applications rely on timed transmissions. Such timed transmissions are usually either produced by human interaction, or responses to certain events. In either case, the application is time-dependent to a certain degree.

- **Hypothesis 2:** *Retransmission mechanisms and congestion control mechanisms have been developed to maximise throughput, and may therefore cause higher retransmission latency when the transported stream is thin.*

Several weaknesses were found in TCP and SCTP retransmission mechanisms for thin-stream transmission. The largest contributing factor to increased latency is the lack of feedback due to high packet IATs in thin streams. When reliability is implemented over unreliable protocols like UDP, the retransmission mechanisms used are very often based on the same principles as for TCP and SCTP. Our observations can therefore be applied generally for thin-stream latency over reliable connections.

- **Hypothesis 3:** *It is possible to adapt existing retransmission and congestion control mechanisms to achieve lower latency for thin streams without jeopardising performance for greedy streams.*

We implemented modifications that were dynamically triggered when the system detected a thin stream. The mechanisms we implemented provided unmodified service to greedy streams while improving latency for thin streams. From our investigations into per-stream fairness, we saw that only the bundling mechanism had any impact on the achieved throughput. We therefore regard the claim to be well-founded. There may be other avenues of investigation, in addition to the ones we have explored, that will lead to reduced latency for time-dependent, thin-stream applications.

- **Hypothesis 4:** *We can take advantage of the thin stream properties to achieve lower delivery latencies for the thin streams.*

Our investigations into this hypothesis resulted in a bundling mechanism that took advantage of the small packet sizes observed in most thin streams. Using an extra measure of redundancy, we were able to significantly reduce latency for a range of time-dependent applications. Here also, other approaches may lead to similar results, and additional limitations can be implemented to reduce the potential overhead for the implemented mechanism.

- **Hypothesis 5:** *Modifications to improve thin-stream latency can be implemented in such a way that unmodified receivers may benefit from them.*

We have, during the development of all our modifications, kept transparency to the receiver as a strict requirement. It is possible to develop other mechanisms or, maybe, new protocols that can address many of the problems we have identified, but we have deemed such wok outside the scope of this thesis. The mechanisms that we have developed can, in practise, be deployed instantly, and be of service to the masses without modifying the receivers. As interactive applications, by being interactive, communicates both ways, the largest benefit would be by having thin-stream support on both sender and receiver. This may be achieved either by modifying existing systems and lobby for standardisation of the changes, or by developing new protocols. It has, however, proved difficult to gain wide acceptance for new protocols, and we have therefore chosen the first alternative.

The term "thin stream" describes a class of streams that differs from greedy streams by having higher packet IAT and smaller packet sizes. Aside from that, thin streams show great variation (as discussed in section 4.3). Some streams dynamically fluctuate between being thin and greedy, other keep the same properties throughout their lifetime and some streams are borderline thin/greedy. Our set of modifications provides significantly lowered latency in many cases, while being less effective in other. We have tried to find ways of limiting the frame of operation for our mechanisms to target the areas where the need is greatest. More effective ways of limiting this frame may exist, though a totally effective way of achieving optimal results without trade-offs may not be possible.

The requirement on thin-stream detection of $in\_transit < 4$ that we used is the most conservative choice. It ensures that the modifications are enabled *only* in situations where fast retransmissions *cannot* occur. There may be good reasons to relax this requirement in order to improve the latency benefits further. For instance, the limit could be increased when heavy loss occurs, which is likely to trigger exponential backoff. With a slightly higher limit for in_transit packets to trigger the thin-stream modifications, a thin stream that will not expand its congestion window further can avoid some of the most punishing latencies.

RDB is active when there are unacknowledged segments and small packet sizes. It is not active if only one segment is on the wire at any given time, nor when the stream is greedy (i.e., when the segments fill up). These inherent limitations ensure that RDB only consumes extra bandwidth when the potential for increased latency is high. For cases in which lowered throughput in streams that compete with RDB-enabled thin streams is unwanted, a trigger could be created to ensure that RDB is only active if IAT > RTT. In this way, RDB would only be active upon loss, yielding latency improvements with a minimum of redundancy. The RDB-activated stream would receive ACKs in situations in which a regular TCP-stream would record loss, so there would still be an effect on other streams with regard to throughput. Such an option could be helpful for streams that fluctuate in bandwidth, but have periods of typical thin-stream activity. In this way, the added redundancy in the transitions between thin and greedy could be reduced significantly.

Another special case for RDB appears for applications whose IAT is low, but not low enough that the packet sizes reach the MTU size. In this case, our measurements show high redundancy due to massive bundling (especially if the bundling is combined with high RTTs). In order to improve latency while reducing this redundancy, a trigger could be implemented that disables RDB if a certain limit is reached ($in\_transit > limit$). In such cases, it would also be possible to enable bundling with only a subset of the packets that are sent.

The performance of different operating systems when transmitting thin streams should be evaluated for an even broader range of parameters. This could provide insights into the cases that suffer the most for the different systems, and help to suggest the system, or system configuration, that would provide the best service for special thin-stream scenarios. It may also help to suggest additional adaptations of the thin-stream mechanisms for each special case.

One of the aspects of applying a dynamic trigger for our mechanisms is that streams may fluctuate between greedy and thin. Such fluctuating streams may suffer higher latencies if loss occurs in the transition between thin and greedy. There may also be effects pertaining to fairness, as for instance the RDB mechanism bundles up to the MSS for a stream going from thin to greedy. The effects of such fluctuating streams when our modifications are applied may therefore be subjected to additional experimentation.

Our experiments to learn about the effects of our modifications on per-stream fairness have focused on thin streams competing with greedy streams, and one-on-one cases (thin stream with gradually decreasing packet IAT). A wider range of test parameters (thin/thick streams, IATs, RTTs, fluctuating streams, loss rates) should be evaluated to get more information about the effects in different scenarios. We should also perform experiments on the number of streams, modified or unmodified, that can share a bottleneck of a certain capacity before loss becomes too severe to support any kind of interactive applications (with or without modifications).

7.4 Future work

During our work on the topic of thin streams over reliable transport, we investigated a wide range of options for increasing the latency for such interactive streams. We performed experiments to evaluate the aspects of our mechanisms that we found most relevant to the subject matter. There are, however, unexplored avenues of investigation. In this section, we outline some of the topics that may extend the work presented in this thesis.

In section 7.3, we discussed alternative ways of identifying thin streams and triggering mechanisms. A possible future expansion of the work reported herein may be to chart which thin-stream limit gives a reasonable tradeoff between latency reduction and redundancy. Experiments could also be performed to find ways of limiting the activation of RDB to the cases where redundancy is well balanced against reduced latency.

Alternative ways to achieve similar results to the presented modifications can be devised. For example, all segments (packets) could be sent twice with a given delay between transmissions. The delay would need to be calculated so that the chance that the second segment is lost within the same burst of losses as the first is minimised, while maximising the chance that the second segment is delivered and ACKed before a retransmission by timeout occurs. In order to keep the level of redundancy low, this mechanism would also have to be limited to thin streams.

TCP Vegas should, if implemented as specified with trusted, fine-grained timers, be able to handle some of the latency-issues that we have identified for time-dependent thin-streams. A future approach should be to validate such an implementation, and compare the results to the modifications described in this thesis.

Finally, as described in section 5.1.1, we have evaluated several simulators for thin-stream experiments. None of the tested alternatives could provide us with a satisfactory solution within the time and resources available. To achieve insights into the effects of our modifications in complex systems larger than we practically can perform experiments on, a simulator should be developed that can cope with both fairness and latency issues as well as handle varying segment sizes.

Bibliography

[1] netem. http://www.linuxfoundation.org/en/Net:Netem, July 2008.

[2] Skype, March 2008. http://www.skype.com.

[3] The adaptive communication environment (ace). http://www.cse.wustl.edu/ schmidt/ACE.html, August 2009.

[4] BZFlag, September 2009. http://bzflag.org.

[5] Clanlib game sdk. http://www.clanlib.org/, August 2009.

[6] Collaborative adaptive sensing of the atmosphere (CASA), Aug 2009. http://www.casa.umass.edu/.

[7] Hawk network library (hawknl). http://www.hawksoft.com/hawknl/, August 2009.

[8] Quazal net-z advanced distributed game state engine. http://www.quazal.com/en/products/net-z/net-z, August 2009.

[9] Rakenet cross-platform c++ game networking engine. http://www.jenkinssoftware.com/, August 2009.

[10] Replicanet multiplatform connectivity. http://www.replicanet.com/, August 2009.

[11] Simple directmedia layer (sdl). http://www.libsdl.org/, August 2009.

[12] Steve's portable game library (plib). http://plib.sourceforge.net/, August 2009.

[13] The Linux Kernel Stream Control Transmission Protocol (lksctp) project, October 2009. http://lksctp.sourceforge.net/.

[14] Udt udp-based data transfer. http://udt.sf.net, August 2009.

[15] Zoidcom automated networking system. http://www.zoidcom.com/, August 2009.

[16] M. Allman, H. Balakrishnan, and S. Floyd. Enhancing TCP's Loss Recovery Using Limited Transmit. RFC 3042 (Proposed Standard), January 2001.

[17] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control . RFC 2581 (Proposed Standard), April 1999. Updated by RFC 3390.

[18] AT Internet Institute. Internet users equipment. http://www.atinternet-institute.com/, September 2009.

[19] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. In *SIGMETRICS '98/PERFORMANCE '98: Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 151–160, New York, NY, USA, 1998. ACM Press.

[20] Vitor Basto and Vasco Freitas. SCTP extensions for time sensitive traffic. In *Proceedings of the International Network Conference (INC)*, July 2005.

[21] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Service. RFC 2475 (Informational), December 1998. Updated by RFC 3260.

[22] Inc Blizzard Entertainment. World of Warcraft. *http://www.worldofwarcraft.com/, January*, 2008.

[23] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Standard), October 1989. Updated by RFC 1349.

[24] R. Braden, D. Clark, and S. Shenker. Integrated Services in the Internet Architecture: an Overview. RFC 1633 (Informational), June 1994.

[25] Lawrence S. Brakmo, Sean W. O'Malley, and Larry L. Peterson. TCP Vegas: new techniques for congestion detection and avoidance. In *Proceedings of the ACM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 24–35. ACM Press, 1994.

[26] Rob Brennan and Thomas Curran. SCTP congestion control: Initial simulation studies. In *Proc. of the International Teletraffic Congress (ITC 17)*, September 2001.

[27] Bungie. Halo 3. http://halo.xbox.com/halo3/, September 2009.

[28] Carlo Caini and Rosario Firrincelli. TCP hybla: a TCP enhancement for heterogeneous networks. *International journal of satellite communications and networking*, 22(5):547–566, 2004.

[29] K. Cardoso and J. de Rezende. HTTP traffic modeling: Development and application, 2002.

[30] Claudio Casetti, Mario Gerla, Saverio Mascolo, M. Y. Sanadidi, and Ren Wang. TCP Westwood: end-to-end congestion control for wired/wireless networks. *Wireless Network*, 8(5):467–479, 2002.

[31] M. Claypool. The effect of latency on user performance in real-time strategy games. *Elsevier Computer Networks*, 49(1):52–70, September 2005.

[32] Mark Claypool and Kajal Claypool. Latency and player actions in online games. *Communications of the ACM*, 49(11):40–45, November 2005.

[33] Clip2. The bitterrent protocol specification. http://www.bittorrent.org/beps/-bep_0003.html.

[34] L. Coene and J. Pastor-Balbas. Telephony Signalling Transport over Stream Control Transmission Protocol (SCTP) Applicability Statement. RFC 4166 (Informational), February 2006.

[35] Cube SourceForge Project. Cube. http://www.cubeengine.com/, May 2007.

[36] Hannes Ekström and Reiner Ludwig. The peak-hopper: A new end-to-end retransmission timer for reliable unicast transport. In *INFOCOM*, 2004.

[37] The enet project. enet website. *http://enet.cubik.org/, July*, 2009xs.

[38] K. Evensen, A. Petlund, C. Griwodz, and P. Halvorsen. Redundant bundling in TCP to reduce perceived latency for time-dependent thin streams. *Communications Letters, IEEE*, 12(4):324–326, April 2008.

[39] Kristian Riktor Evensen. Improving TCP for time-dependent applications. Master's thesis, Department of Informatics, University of Oslo, Oslo, Norway, May 2008.

[40] S. Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649 (Experimental), December 2003.

[41] S. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782 (Proposed Standard), April 2004.

[42] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC 2883 (Proposed Standard), July 2000.

[43] Funcom. Age of conan. http://www.ageofconan.com/, September 2009.

[44] Funcom. Anarchy online. http://www.anarchy-online.com/, September 2009.

[45] Ashvin Goel, Charles Krasic, and Jonathan Walpole. Low-latency adaptive streaming over tcp. *ACM Trans. Multimedia Comput. Commun. Appl.*, 4(3):1–20, 2008.

[46] Asvin Goel, Charles Krasic, Kang Li, and Jonathan Walpole. Supporting low latency TCP-based media streams. In *Proceedings of the IEEE International Workshop on Quality of Service (IWQoS)*, pages 193–203, May 2002.

[47] Google. Google Android for mobile devices. http://www.android.com/, September 2009.

[48] Luigi A. Grieco and Saverio Mascolo. Performance evaluation and comparison of Westwood+, New Reno, and Vegas TCP congestion control. *ACM Computer Communication Review*, 34(2):25–38, 2004.

[49] Karl-Johan Grinnemo and Anna Brunstrom. Performance of SCTP-controlled failovers in M3UA-based SIGTRAN networks. In *Proc. of the Advanced Simulation Technologies Conference (ASTC)*, April 2004.

[50] Carsten Griwodz and Pål Halvorsen. The fun of using TCP for an MMORPG. In *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 1–7. ACM Press, May 2006.

[51] Carsten Griwodz, Knut-Helge Vik, and Pål Halvorsen. Multicast tree reconfiguration in distributed interactive applications. In *Proceedings of the International Conference (NIME)*, pages 1219 – 1223, January 2006.

[52] Yunhong Gu and Robert L. Grossman. UDT: UDP-based Data Transfer for High-Speed Wide Area Networks. *Computer Networks (Elsevier)*, 51(7), May 2007.

[53] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *SIGOPS Oper. Syst. Rev.*, 42(5):64–74, 2008.

[54] Szabolcs Harcsik, Andreas Petlund, Carsten Griwodz, and Pål Halvorsen. Latency evaluation of networking mechanisms for game traffic. In *Proceedings of the Workshop on Network and System Support for Games (NETGAMES)*, pages 129–134, September 2007.

[55] Mahbub Hassan and Danilkin Fiodor Alekseevich. Variable packet size of ip packets for voip transmission. In *Proceedings of the IASTED International Conference conference on Internet and Multimedia Systems and Applications (IMSA)*, pages 136–141. ACTA Press, 2006.

[56] Sofiane Hassayoun and David Ros. Improving application layer latency for reliable thin-stream game traffic. In *Accepted for the 34th Annual IEEE Conference on Local Computer Networks (LCN)*, October 2009.

[57] P. Hurtig and A. Brunstrom. Packet loss recovery of signaling traffic in sctp. In *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS0́7)*, San Diego, California, July 2007.

[58] IEEE. *IEEE 802.3-2008 - Section One*, 2008.

[59] International Telecommunication Union (ITU-T). One-way Transmission Time, ITU-T Recommendation G.114, 2003.

[60] Van Jacobson. Congestion avoidance and control. In *ACM SIGCOMM '88*, pages 314–329, Stanford, CA, August 1988.

[61] Phil Karn and Craig Partridge. Improving round-trip time estimates in reliable transport protocols. pages 2–7, 1988.

[62] Tom Kelly. Scalable tcp: improving performance in highspeed wide area networks. *SIGCOMM Comput. Commun. Rev.*, 33(2):83–91, 2003.

[63] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), March 2006.

[64] Charles Krasic, Kang Li, and Jonathan Walpole. The case for streaming multimedia with tcp. In *Proceedings of the International Workshop on Interactive Distributed Multimedia Systems and Telecommunication Services (IDMS)*, September 2001.

[65] Sourabh Ladha, Stephan Baucke, Reiner Ludwig, and Paul D. Amer. On making SCTP robust to spurious retransmissions. *ACM Computer Communication Review*, 34(2):123–135, 2004.

[66] D. Leith and R. Shorten. H-TCP: TCP congestion control for high bandwidth-delay product paths, June 2005. `http://www.hamilton.ie/net/draft-leith-tcp-htcp-00.txt`.

[67] R. Ludwig and K. Sklower. The Eifel retransmission timer. *SIGCOMM Comput. Commun. Rev.*, 30(3):17–27, 2000.

[68] H. Lundqvist and G. Karlsson. Tcp with end-to-end fec. In *Communications, 2004 International Zurich Seminar on*, pages 152–155, 2004.

[69] Chris Majewski, Carsten Griwodz, and Pål Halvorsen. Translating latency requirements into resource requirements for game traffic. In *Proceedings of the International Network Conference (INC)*, July 2006.

[70] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgement Options. RFC 2018 (Proposed Standard), October 1996.

[71] Matthew Mathis and Jamshid Mahdavi. Forward acknowledgement: refining TCP congestion control. In *Proceedings of the ACM International Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 281–291. ACM Press, 1996.

[72] Microsoft. The compound tcp for high-speed and long distance networks. http://research.microsoft.com/en-us/projects/ctcp/, September 2009.

[73] Microsoft. New networking features in windows server 2008 and windows vista. http://technet.microsoft.com/en-us/library/bb726965.aspx, September 2009.

[74] Dimitrios Miras, Martin Bateman, and Saleem Bhatti. Fairness of high-speed tcp stacks. In *AINA '08: Proceedings of the 22nd International Conference on Advanced Information Networking and Applications*, pages 84–92, Washington, DC, USA, 2008. IEEE Computer Society.

[75] Amit Mondal and Aleksandar Kuzmanovic. Removing exponential backoff from tcp. *SIGCOMM Comput. Commun. Rev.*, 38(5):17–28, 2008.

[76] J. Nagle. Congestion control in IP/TCP internetworks. RFC 896, January 1984.

[77] Espen Søgård Paaby. Evaluation of TCP retransmission delays. Master's thesis, Department of Informatics, University of Oslo, Oslo, Norway, May 2006.

[78] Wladimir Palant, Carsten Griwodz, and Pål Halvorsen. Consistency requirements in multiplayer online games. In Stephane Natkin Adrian David Cheok, Yutaka Ishibashi and Keiichi Yasumoto, editors, *Network & System Support for Games (NetGames 2006)*, pages 1–4. ACM Press, 2006.

[79] Wladimir Palant, Carsten Griwodz, and Pål Halvorsen. Evaluating dead reckoning variations with a multi-player game simulator. In *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, pages 20–25, May 2006.

[80] Wladimir Palant, Carsten Griwodz, and PÃěl Halvorsen. Gls: Simulator for online multiplayer games. In Kenji Mase, editor, *ACM Multimedia (MM 2006)*, pages 805–806. ACM Press, 2006.

[81] Christina Parsa and J. J. Garcia-Luna-Aceves. Improving TCP congestion control over internets with heterogeneous transmission media. In *International Conference on Network Protocols (ICNP)*, pages 213–221, November 1999.

[82] V. Paxson and M. Allman. Computing TCP's Retransmission Timer. RFC 2988 (Proposed Standard), November 2000.

[83] Jon Pedersen. Evaluation of SCTP retransmission delays. Master's thesis, Department of Informatics, University of Oslo, Oslo, Norway, May 2006.

[84] Jon Pedersen, Carsten Griwodz, and Pål Halvorsen. Considerations of SCTP retransmission delays for thin streams. In *Proceedings of the IEEE Conference on Local Computer Networks (LCN)*, pages 1–12, November 2006.

[85] Andreas Petlund, Paul Beskow, Jon Pedersen, Espen Søgård Paaby, Carsten Griwodz, and Pål Halvorsen. Improving sctp retransmission delays for time-dependent thin streams. *Springer's Multimedia Tools and Applications*, Special Issue on Massively Multiuser Online Gaming Systems and Applications, 2009.

[86] Andreas Petlund, Kristian Evensen, , Carsten Griwodz, and Pål Halvorsen. Improving application layer latency for reliable thin-stream game traffic. In *Proceedings of the Workshop on Network and System Support for Games (NETGAMES)*, pages 91–98, October 2008.

[87] Andreas Petlund, Kristian Evensen, Carsten Griwodz, and Pål Halvorsen. TCP mechanisms for improving the user experience for time-dependent thin-stream applications. In Chun Tung Chou Ehab Elmallah, Mohamed Younis, editor, *The 33rd Annual IEEE Conference on Local Computer Networks (LCN)*. IEEE, 2008.

[88] Andreas Petlund, Kristian R Evensen, Carsten Griwodz, and Pål Halvorsen. Latency reducing tcp modifications for thin-stream interactive applications. In W. Stief, editor, *UpTimes - Magazine of the German Unix User Group (Proceedings of Linux Kongress 2008)*, number 2, pages 150–154, Bachemer Str. 12, 50931 Köln, 2008. German Unix User Group, German Unix User Group.

[89] Andreas Petlund, Kristian R Evensen, Carsten Griwodz, and Pål Halvorsen. Tcp enhancements for interactive thin-stream applications. In Carsten Griwodz and Lars Wolf,

editors, *Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2008)*, pages 127–128. ACM, 2008. short paper and demo.

[90] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[91] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFC 3168.

[92] Batu Sat and Benjamin W. Wah. Playout scheduling and loss-concealments in voip for optimizing conversational voice communication quality. In *Proceedings of the ACM International Multimedia Conference (ACM MM)*, pages 137–146, October 2007.

[93] SIGTRAN. Ietf signaling transport workgroup. http://tools.ietf.org/wg/sigtran/, July 2009.

[94] Spotify. Spotify online music service. http://www.spotify.com/, July 2009.

[95] R. Stewart. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), September 2007.

[96] R. Stewart, M. Ramalho, Q. Xie, M. Tuexen, and P. Conrad. Stream Control Transmission Protocol (SCTP) Partial Reliability Extension. RFC 3758 (Proposed Standard), May 2004.

[97] R. Stewart, Q. Xie, K. Morneault, C. Sharp, H. Schwarzbauer, T. Taylor, I. Rytina, M. Kalla, L. Zhang, and V. Paxson. Stream Control Transmission Protocol. RFC 2960 (Proposed Standard), October 2000. Updated by RFC 3309.

[98] SIMREAL technology. NCTUns network simulator. http://nsl10.csie.nctu.edu.tw/, September 2009.

[99] The Clacoons. "Kålfis i Natten". http://www.youtube.com/watch?v=rGqeaG0g92s, October 1997. Toneveld performance.

[100] The Entertainment Software Association. ESAs 2006 essential facts about the computer and video game industry, January 2008.

[101] Knut-Helge Vik. Game state and event distribution using proxy technology and application layer multicast. In *Proceedings of the ACM International Multimedia Conference (ACM MM)*, pages 1041–1042, 2005.

[102] Knut-Helge Vik, Carsten Griwodz, and Pål Halvorsen. Applicability of group communication for increased scalability in MMOGs. In *Proceedings of the Workshop on Network and System Support for Games (NETGAMES)*, Singapore, October 2006. ACM Press.

[103] Knut-Helge Vik, Carsten Griwodz, and Pål Halvorsen. Dynamic group membership management for distributed interactive applications. In *Proceedings of the IEEE Conference on Local Computer Networks (LCN)*, pages 141–148, October 2007.

[104] Knut-Helge Vik, Carsten Griwodz, and Pål Halvorsen. On the influence of latency estimation on dynamic group communication using overlays, *(to appear)*. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking (MMCN)*, San Jose, CA, USA, January 2009.

[105] Knut-Helge Vik, Pål Halvorsen, and Carsten Griwodz. Constructing low-latency overlay networks: Tree vs. mesh algorithms. In *Proceedings of the IEEE Conference on Local Computer Networks (LCN)*, October 2008.

[106] Knut-Helge Vik, Pål Halvorsen, and Carsten Griwodz. Evaluating steiner tree heuristics and diameter variations for application layer multicast. *Elsevier Computer Networks*, 52(15):2872–2893, October 2008.

[107] Knut-Helge Vik, Pål Halvorsen, and Carsten Griwodz. Multicast tree diameter for dynamic distributed interactive applications. In *Proceedings of the Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, pages 1597–1605, April 2008.

[108] Bing Wang, Jim Kurose, Prashant Shenoy, and Don Towsley. Multimedia streaming via TCP: an analytic performance study. In *MULTIMEDIA '04: Proceedings of the 12th annual ACM international conference on Multimedia*, pages 908–915, New York, NY, USA, 2004. ACM.

[109] Bing Wang, Jim Kurose, Prashant Shenoy, and Don Towsley. Multimedia streaming via tcp: An analytic performance study. *ACM Trans. Multimedia Comput. Commun. Appl.*, 4(2):1–22, 2008.

[110] Yi Wang, Guohan Lu, and Xing Li. A study of internet packet reordering. In *ICOIN*, pages 350–359, 2004.

[111] B. S. Woodcock. An analysis of mmog subscription growth, April 2009.

[112] www.sctp.org. Sctp implementations, September 2009. `http://www.sctp.org/implementations.html`.

[113] Lisong Xu, Khaled Harfoush, and Injong Rhee. Binary increase congestion control for fast long-distance networks. In *Proceedings of the Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2004.

[114] Michael Zink, David Westbrook, Sherief Abdallah, Bryan Horling, Vijay Lakamraju, Eric Lyons, Victoria Manfredi, Jim Kurose, and Kurt Hondl. Meteorological command and control: An end-to-end architecture for a hazardous weather detection sensor network. In *Proceedings of the ACM Workshop on End-to-End, Sense-and-Respond Systems, Applications, and Services (EESR)*, pages 37–42, Seattle, WA, June 2005.

Appendix A

List of abbreviations

ACK	Acknowledgement
AIMD	Additive increase / multiplicative decrease
AO	Anarchy Online
AoC	Age of Conan
BDP	Bandwidth-delay product
B	Bytes
Bps	Bytes per second
CDF	Cumulative distribution function
cumACK	Cumulative acknowledgement
cwnd	Congestion window
DCCP	Datagram congestion control protocol
DSACK	Duplicate selective acknowledgement
dupACK	Duplicate acknowledgement
EFR	Early fast retransmit
FACK	Forward acknowledgement
FPS	First-person shooter (game)
FTP	File transfer protocol
HTTP	Hypertext transfer protocol
IAT	Interarrival time
ICMP	Internet control message protocol
IP	Internet protocol
ISP	Internet service provider
kbps	Kilo-bits per second
lksctp	Linux kernel SCTP
LT	Linear timeouts (thin-stream modification)

mFR	Modified fast retransmit (thin-stream modification)
MMOG	Massively multiplayer online game
ms	Milliseconds
MSS	Maximum segment size
OS	Operating system
OWD	One-way delay
p2p	Peer-to-peer
pps	Packets per second
QoE	Quality of experience
QoS	Quality of service
RDB	Redundant data bundling (thin-stream modification)
RDP	Remote desktop protocol
RFC	Request for comments
RPG	Role-playing game
RTO	Retransmission timeout
RTO$_{min}$	Minimum retransmission timeout
RTS	Real-time strategy (game)
RTTVAR	RTT variance
RTT	Round trip time
SACK	Selective acknowledgement
SCTP	Stream control transmission protocol
SKB	Linux TCP segment control block
SRTT	Smoothed RTT
SSH	Secure shell
ssthresh	Slow-start threshold
TCP	Transmission control protocol
tohd	Transmission overhead
TSN	Transmission sequence number
UDP	User datagram protocol
VNC	Virtual network computing
VoIP	Voice over IP
WiC	World in Conflict
WoW	World of Warcraft
wu	World unit (BZFlag distance metric)

Appendix B

Tools

- analyzeTCP and analyzeSCTP: Used to analyse the ACK delay and delivery delay from packet traces. For ACK delay, only a sender-side trace is needed. for delivery delay sender and receiver-side traces are needed.

- tracepump: Logs the packet IATs and packet sizes of original, not retransmitted, packets from a trace file. It opens a connection to a listening server and replays the exact data patterns (IAT and packet sizes) over the network.

- streamzero: Creates a connection to a listening server and produces a thin stream. The thin stream parameters (packet size and IAT) can be specified for each stream that is created.

- interarrival: Gathers information about packet sizes and IATs from a trace file. Used to create the statistics presented in table 2.1.

- http-server/http-client: Used to create cross traffic for creating loss on a bandwidth-limited bottleneck with a tail-dropping queue.

- scripts: A wide range of scripts for setting up batches of tests, generating statistics and plots.

- tcpdump: Used to capture packets and create traces for analysis.

- tc / netem: Used to create bandwidth limitation, delay and loss.

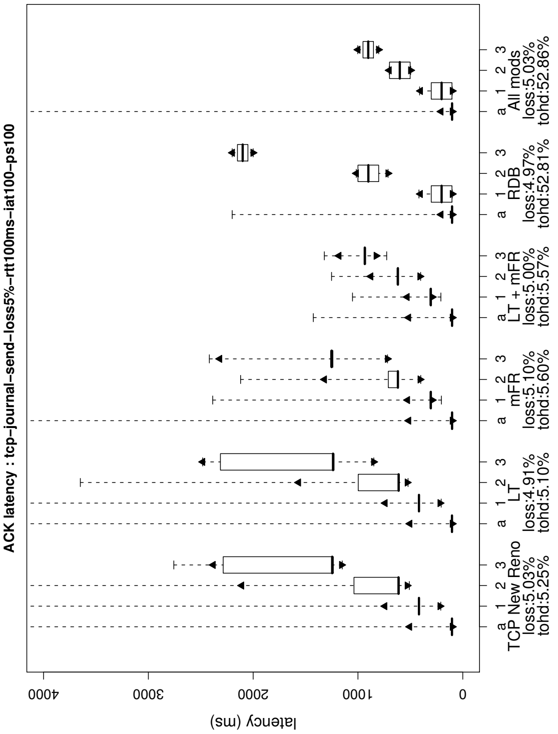
- sctp_perf: Used to create custom thin stream traffic for SCTP.

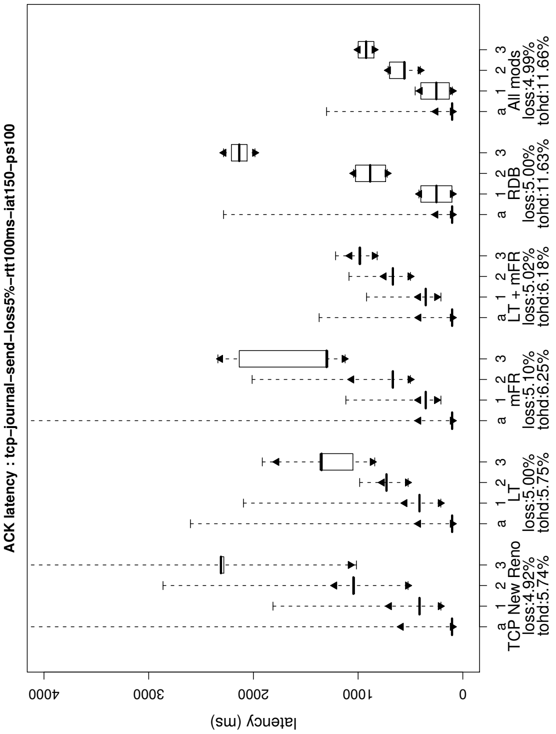
Appendix C

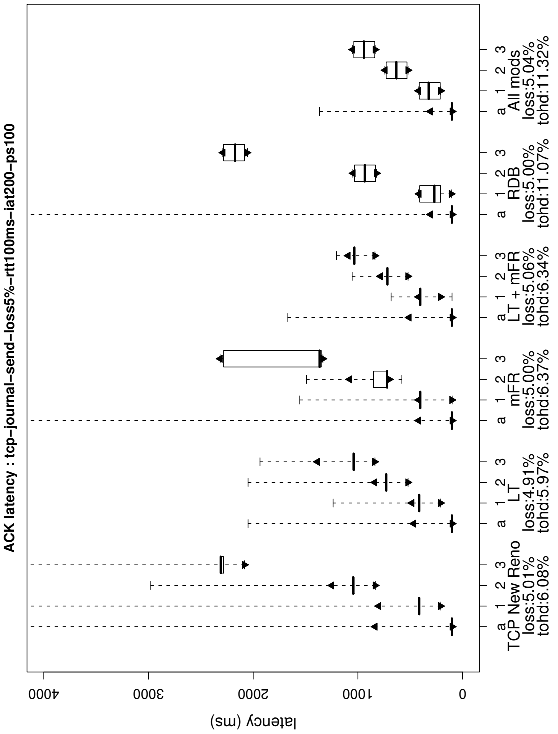
Complete set of boxplots from TCP laboratory experiments

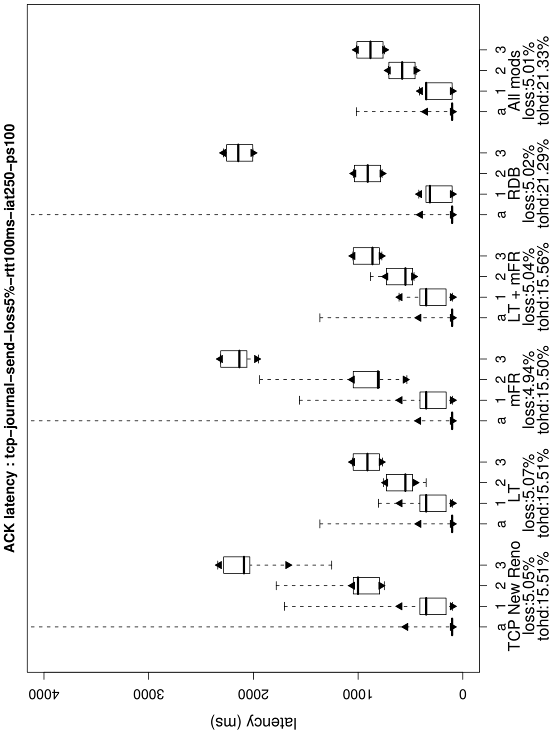
This appendix contains the whole range of boxplots from our laboratory experiments using the thin-stream TCP modifications. This is an extension of the selected statistics that are presented and discussed in section 5.2.

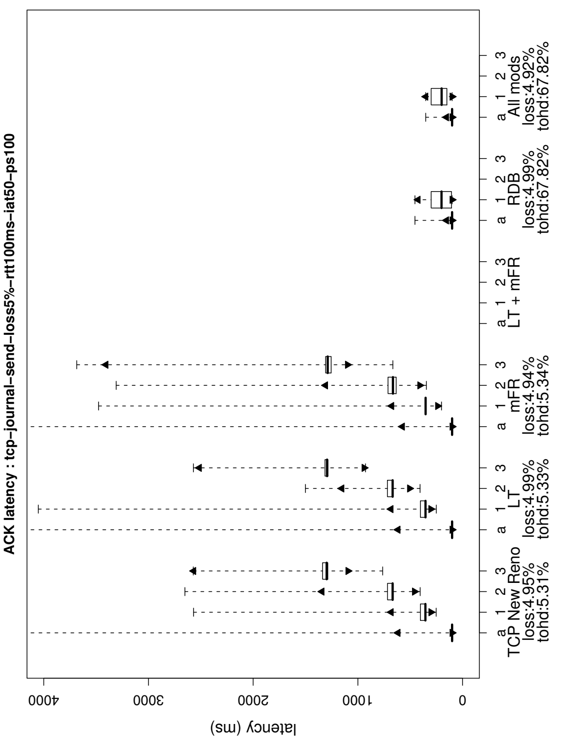
C.1 Uniform loss

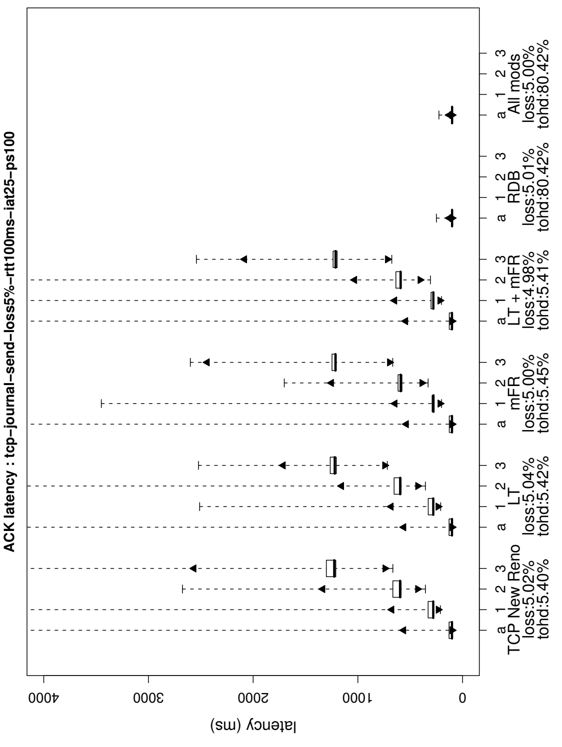

ACK latency : tcp−journal−send−loss5%−rtt100ms−iat100−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:5.03%
tohd:5.25%
a 1 2 3
LT
loss:4.91%
tohd:5.10%
a 1 2 3
mFR
loss:5.10%
tohd:5.60%
a 1 2 3
LT + mFR
loss:5.00%
tohd:5.57%
a 1 2 3
RDB
loss:4.97%
tohd:52.81%
a 1 2 3
All mods
loss:5.03%
tohd:52.86%

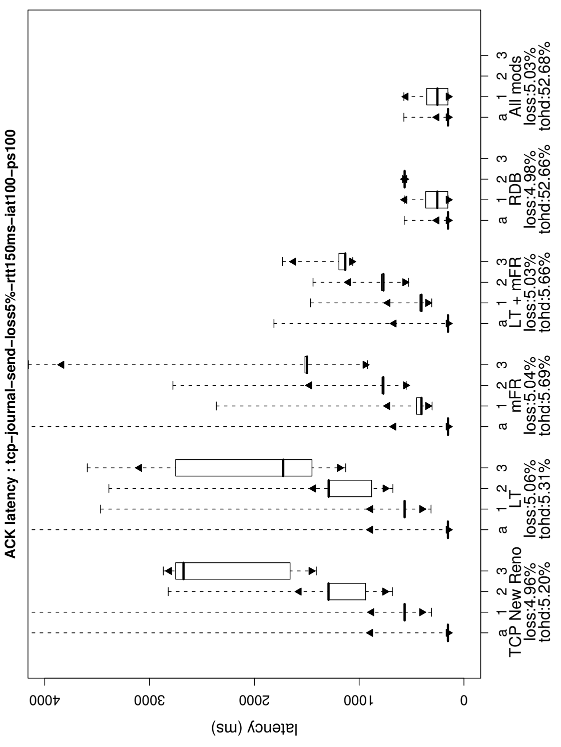

ACK latency : tcp−journal−send−loss5%−rtt100ms−iat150−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.92%
tohd:5.74%
a 1 2 3
LT
loss:5.00%
tohd:5.75%
a 1 2 3
mFR
loss:5.10%
tohd:6.25%
a 1 2 3
LT + mFR
loss:5.02%
tohd:6.18%
a 1 2 3
RDB
loss:5.00%
tohd:11.63%
a 1 2 3
All mods
loss:4.99%
tohd:11.66%

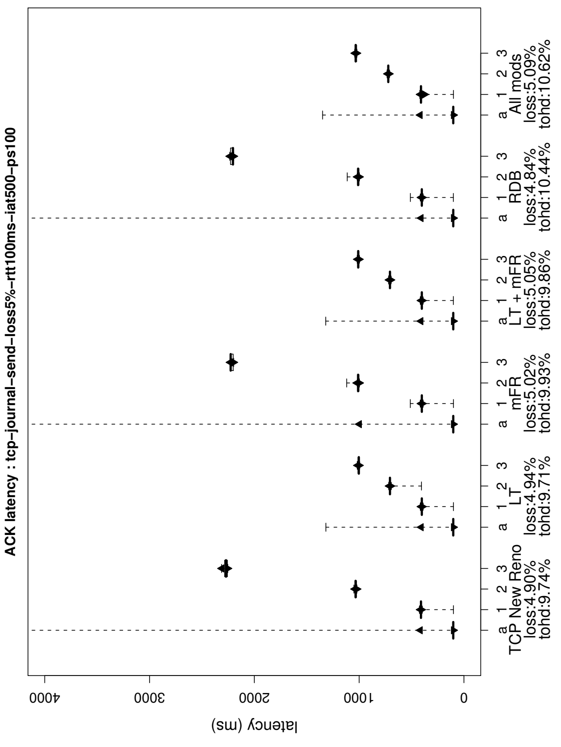

ACK latency : tcp−journal−send−loss5%−rtt100ms−iat200−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:5.01%
tohd:6.08%
a 1 2 3
LT
loss:4.91%
tohd:5.97%
a 1 2 3
mFR
loss:5.00%
tohd:6.37%
a 1 2 3
LT + mFR
loss:5.06%
tohd:6.34%
a 1 2 3
RDB
loss:5.00%
tohd:11.07%
a 1 2 3
All mods
loss:5.04%
tohd:11.32%

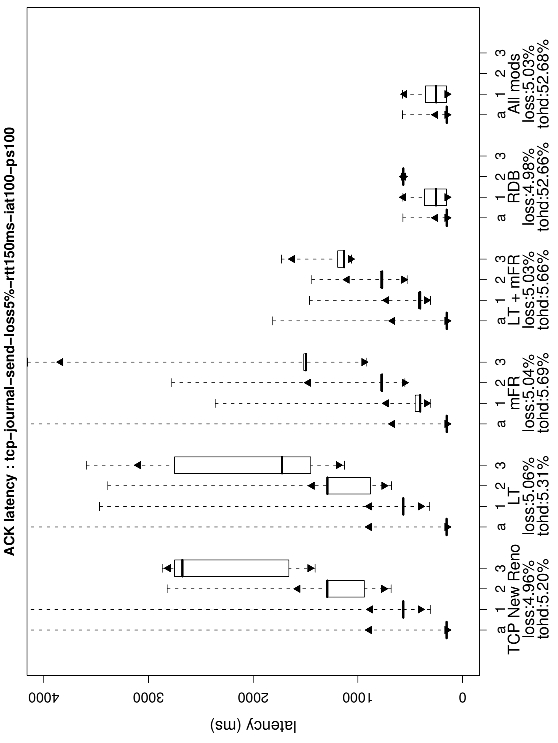

ACK latency : tcp−journal−send−loss5%−rtt100ms−iat250−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:5.05%
tohd:15.51%
a 1 2 3
LT
loss:5.07%
tohd:15.51%
a 1 2 3
mFR
loss:4.94%
tohd:15.50%
a 1 2 3
LT + mFR
loss:5.04%
tohd:15.56%
a 1 2 3
RDB
loss:5.02%
tohd:21.29%
a 1 2 3
All mods
loss:5.01%
tohd:21.33%

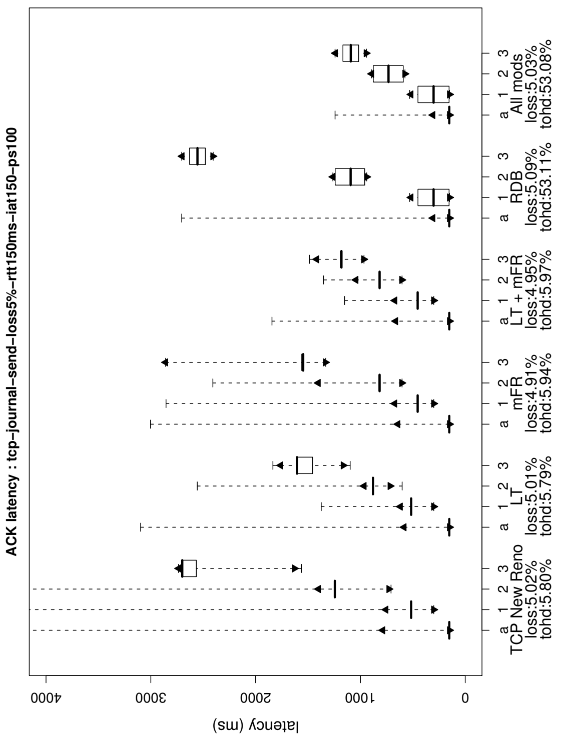

ACK latency : tcp−journal−send−loss5%−rtt100ms−iat50−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.95%
tohd:5.31%
a 1 2 3
LT
loss:4.99%
tohd:5.33%
a 1 2 3
mFR
loss:4.94%
tohd:5.34%
a 1 2 3
LT + mFR
a 1 2 3
RDB
loss:4.99%
tohd:67.82%
a 1 2 3
All mods
loss:4.92%
tohd:67.82%

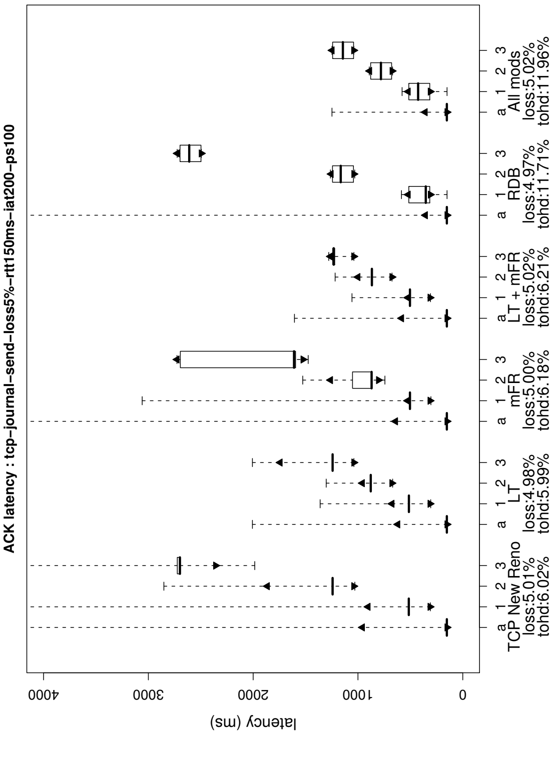

ACK latency : tcp−journal−send−loss5%−rtt100ms−iat25−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:5.02%
tohd:5.40%
a 1 2 3
LT
loss:5.04%
tohd:5.42%
a 1 2 3
mFR
loss:5.00%
tohd:5.45%
a 1 2 3
LT + mFR
loss:4.98%
tohd:5.41%
a 1 2 3
RDB
loss:5.01%
tohd:80.42%
a 1 2 3
All mods
loss:5.00%
tohd:80.42%

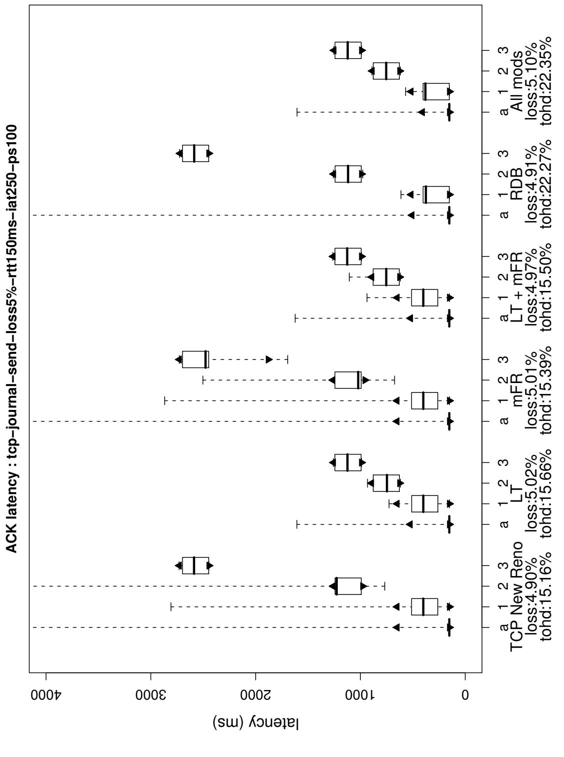

ACK latency : tcp−journal−send−loss5%−rtt150ms−iat100−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.96%
tohd:5.20%
a 1 2 3
LT
loss:5.06%
tohd:5.31%
a 1 2 3
mFR
loss:5.04%
tohd:5.69%
a 1 2 3
LT + mFR
loss:5.03%
tohd:5.66%
a 1 2 3
RDB
loss:4.98%
tohd:52.66%
a 1 2 3
All mods
loss:5.03%
tohd:52.68%

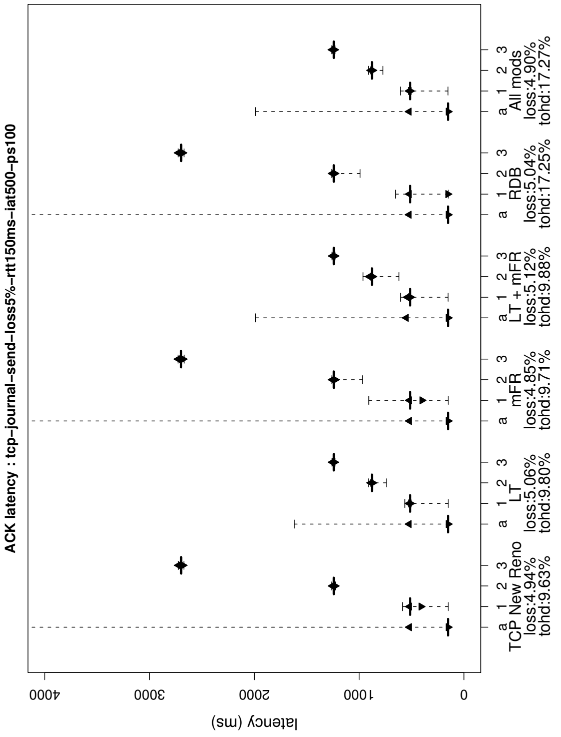

ACK latency : tcp−journal−send−loss5%−rtt100ms−iat500−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.90%
tohd:9.74%
a 1 2 3
LT
loss:4.94%
tohd:9.71%
a 1 2 3
mFR
loss:5.02%
tohd:9.93%
a 1 2 3
LT + mFR
loss:5.05%
tohd:9.86%
a 1 2 3
RDB
loss:4.84%
tohd:10.44%
a 1 2 3
All mods
loss:5.09%
tohd:10.62%

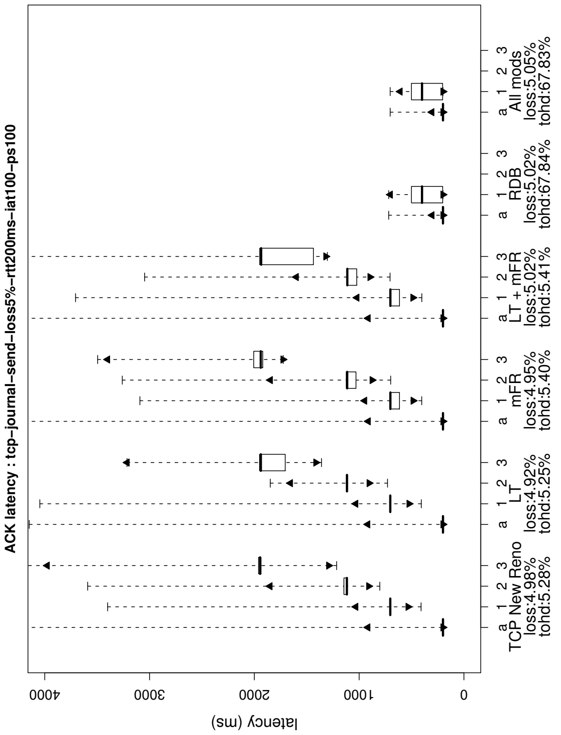

ACK latency : tcp-journal-send-loss5%-rtt150ms-iat100-ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.96%
tohd:5.20%
a 1 2 3
LT
loss:5.06%
tohd:5.31%
a 1 2 3
mFR
loss:5.04%
tohd:5.69%
a 1 2 3
LT + mFR
loss:5.03%
tohd:5.66%
a 1 2 3
RDB
loss:4.98%
tohd:52.66%
a 1 2 3
All mods
loss:5.03%
tohd:52.68%

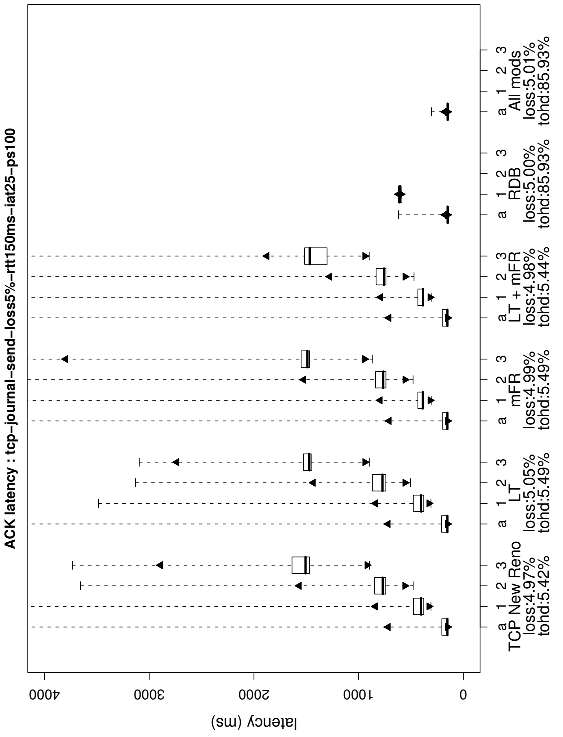

ACK latency : tcp-journal-send-loss5%-rtt150ms-iat150-ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:5.02%
tohd:5.80%
a 1 2 3
LT
loss:5.01%
tohd:5.79%
a 1 2 3
mFR
loss:4.91%
tohd:5.94%
a 1 2 3
LT + mFR
loss:4.95%
tohd:5.97%
a 1 2 3
RDB
loss:5.09%
tohd:53.11%
a 1 2 3
All mods
loss:5.03%
tohd:53.08%

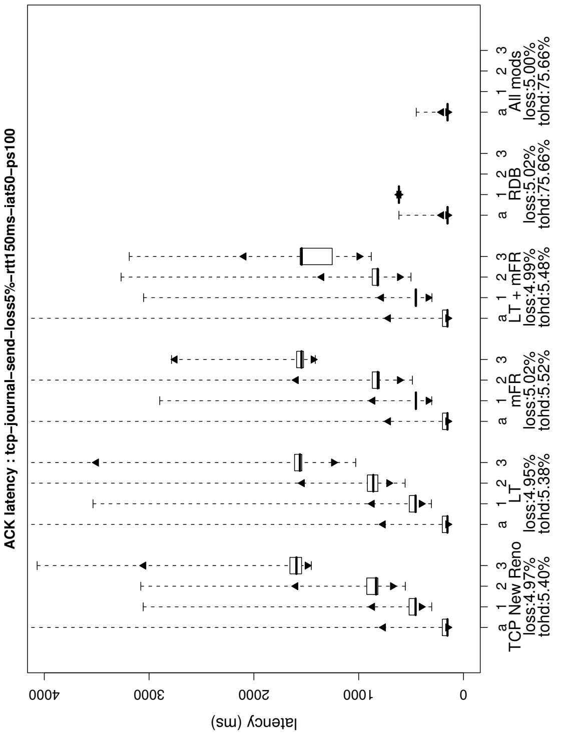

ACK latency : tcp-journal-send-loss5%-rtt150ms-iat200-ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:5.01%
tohd:6.02%
a 1 2 3
LT
loss:4.98%
tohd:5.99%
a 1 2 3
mFR
loss:5.00%
tohd:6.18%
a 1 2 3
LT + mFR
loss:5.02%
tohd:6.21%
a 1 2 3
RDB
loss:4.97%
tohd:11.71%
a 1 2 3
All mods
loss:5.02%
tohd:11.96%

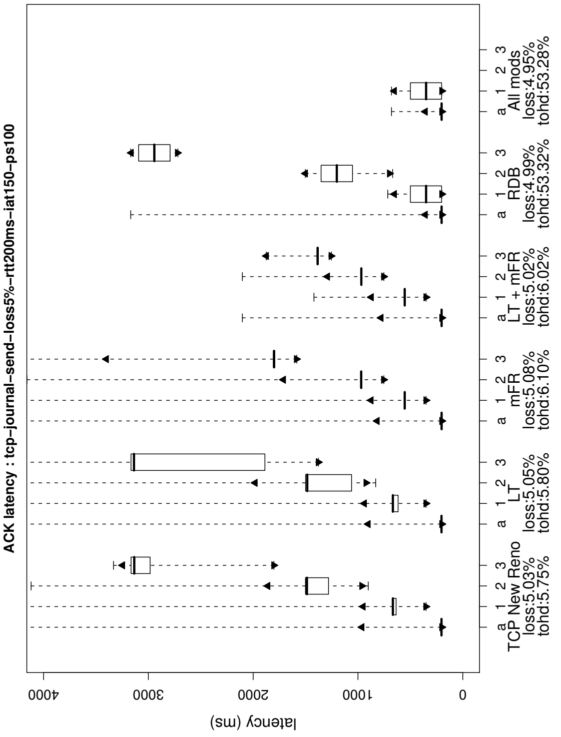

ACK latency : tcp-journal-send-loss5%-rtt150ms-iat250-ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.90%
tohd:15.16%
a 1 2 3
LT
loss:5.02%
tohd:15.66%
a 1 2 3
mFR
loss:5.01%
tohd:15.39%
a 1 2 3
LT + mFR
loss:4.97%
tohd:15.50%
a 1 2 3
RDB
loss:4.91%
tohd:22.27%
a 1 2 3
All mods
loss:5.10%
tohd:22.35%

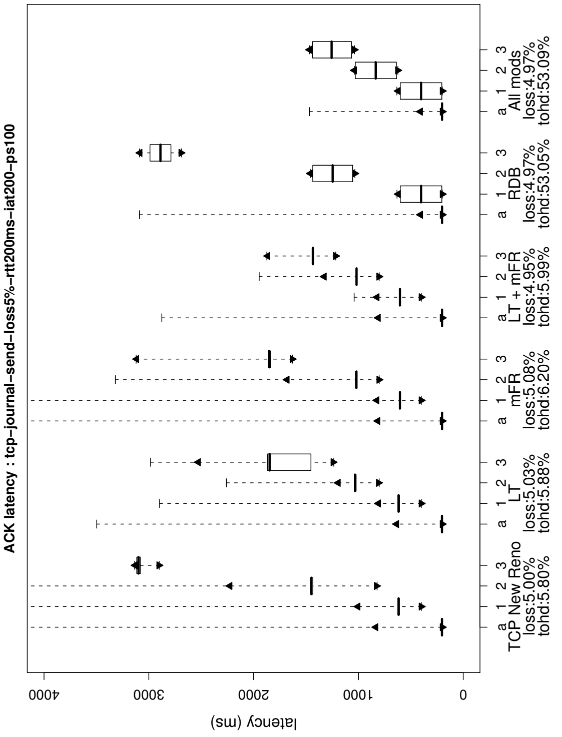

ACK latency : tcp-journal-send-loss5%-rtt150ms-iat500-ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.94%
tohd:9.63%
a 1 2 3
LT
loss:5.06%
tohd:9.80%
a 1 2 3
mFR
loss:4.85%
tohd:9.71%
a 1 2 3
LT + mFR
loss:5.12%
tohd:9.88%
a 1 2 3
RDB
loss:5.04%
tohd:17.25%
a 1 2 3
All mods
loss:4.90%
tohd:17.27%

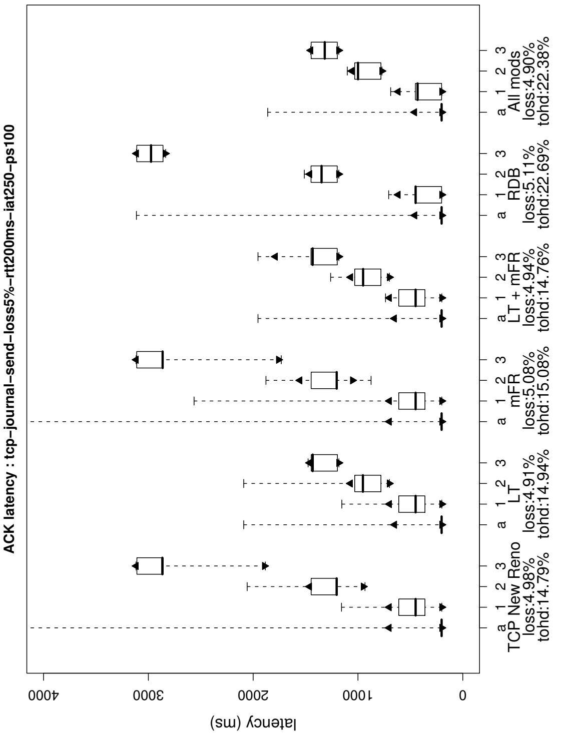

ACK latency : tcp-journal-send-loss5%-rtt200ms-iat100-ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.98%
tohd:5.28%
a 1 2 3
LT
loss:4.92%
tohd:5.25%
a 1 2 3
mFR
loss:4.95%
tohd:5.40%
a 1 2 3
LT + mFR
loss:5.02%
tohd:5.41%
a 1 2 3
RDB
loss:5.02%
tohd:67.84%
a 1 2 3
All mods
loss:5.05%
tohd:67.83%

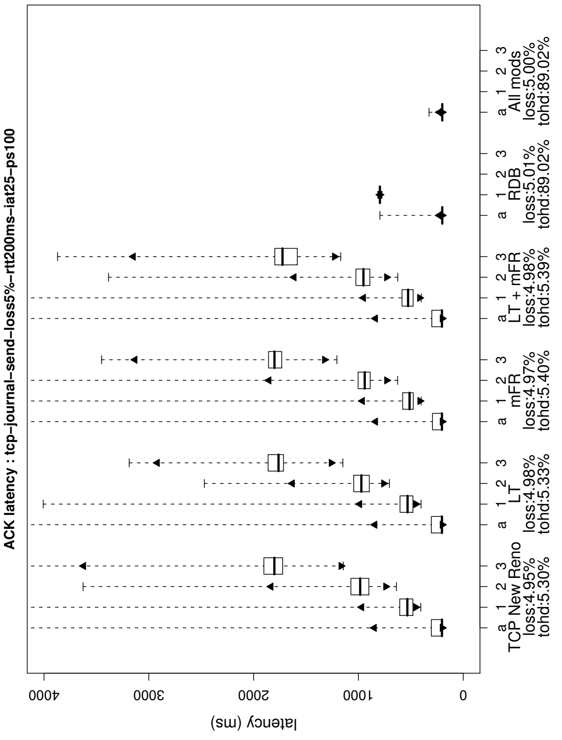

ACK latency : tcp-journal-send-loss5%-rtt150ms-iat25-ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.97%
tohd:5.42%
a 1 2 3
LT
loss:5.05%
tohd:5.49%
a 1 2 3
mFR
loss:4.99%
tohd:5.49%
a 1 2 3
LT + mFR
loss:4.98%
tohd:5.44%
a 1 2 3
RDB
loss:5.00%
tohd:85.93%
a 1 2 3
All mods
loss:5.01%
tohd:85.93%

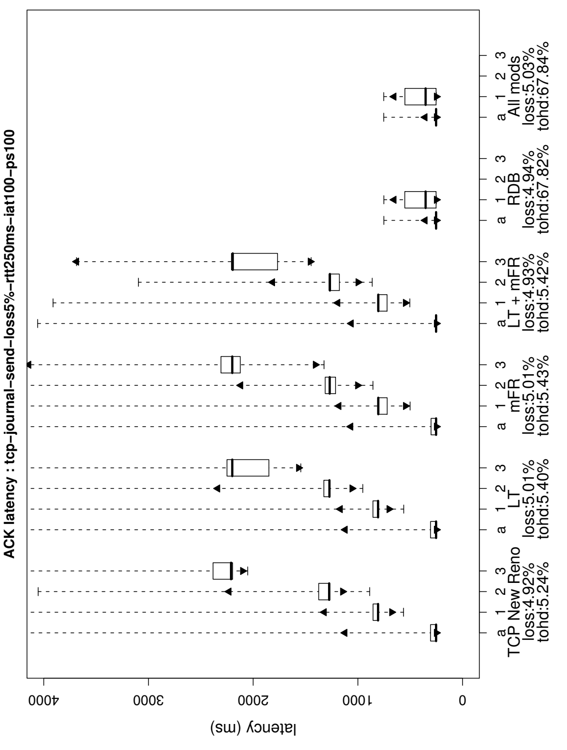

ACK latency : tcp-journal-send-loss5%-rtt150ms-iat50-ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.97%
tohd:5.40%
a 1 2 3
LT
loss:4.95%
tohd:5.38%
a 1 2 3
mFR
loss:5.02%
tohd:5.52%
a 1 2 3
LT + mFR
loss:4.99%
tohd:5.48%
a 1 2 3
RDB
loss:5.02%
tohd:75.66%
a 1 2 3
All mods
loss:5.00%
tohd:75.66%

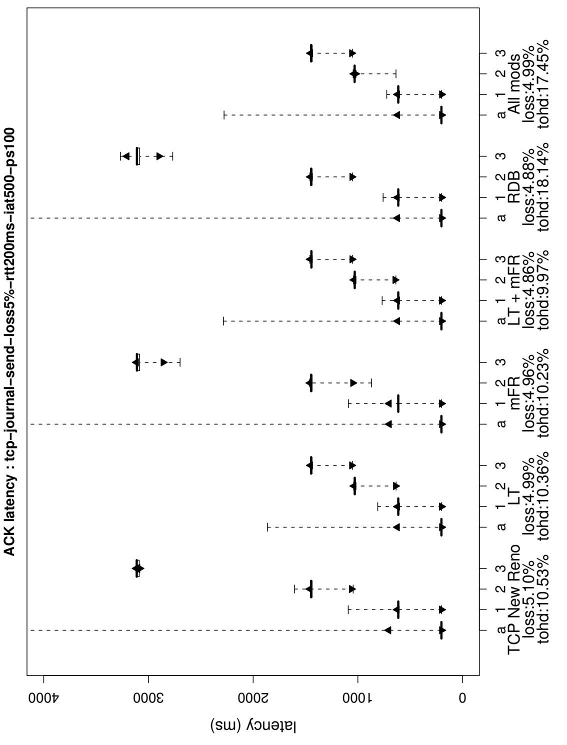

ACK latency : tcp–journal–send–loss5%–rtt200ms–iat150–ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:5.03%
tohd:5.75%
a 1 2 3
LT
loss:5.05%
tohd:5.80%
a 1 2 3
mFR
loss:5.08%
tohd:6.10%
a 1 2 3
LT + mFR
loss:5.02%
tohd:6.02%
a 1 2 3
RDB
loss:4.99%
tohd:53.32%
a 1 2 3
All mods
loss:4.95%
tohd:53.28%

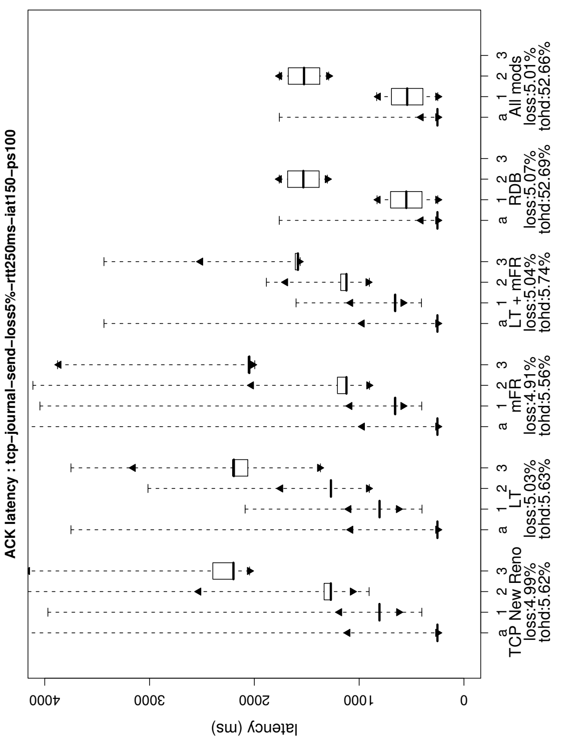

ACK latency : tcp–journal–send–loss5%–rtt200ms–iat200–ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:5.00%
tohd:5.80%
a 1 2 3
LT
loss:5.03%
tohd:5.88%
a 1 2 3
mFR
loss:5.08%
tohd:6.20%
a 1 2 3
LT + mFR
loss:4.95%
tohd:5.99%
a 1 2 3
RDB
loss:4.97%
tohd:53.05%
a 1 2 3
All mods
loss:4.97%
tohd:53.09%

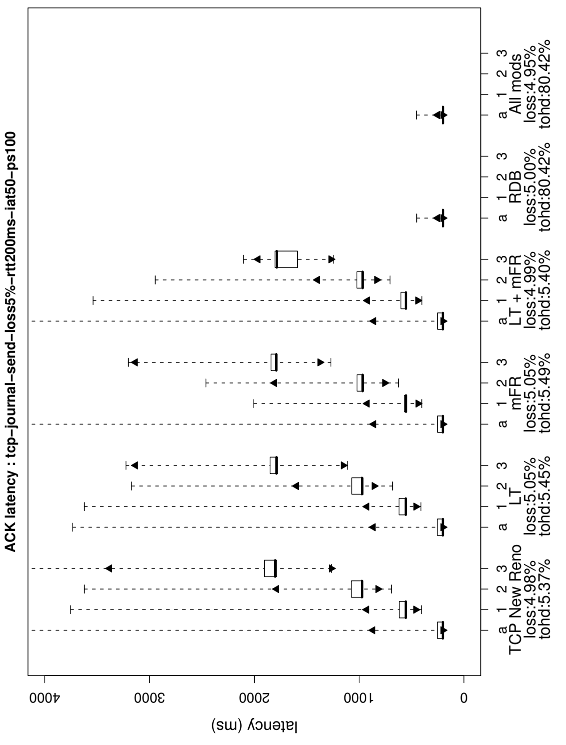

ACK latency : tcp–journal–send–loss5%–rtt200ms–iat250–ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.98%
tohd:14.79%
a 1 2 3
LT
loss:4.91%
tohd:14.94%
a 1 2 3
mFR
loss:5.08%
tohd:15.08%
a 1 2 3
LT + mFR
loss:4.94%
tohd:14.76%
a 1 2 3
RDB
loss:5.11%
tohd:22.69%
a 1 2 3
All mods
loss:4.90%
tohd:22.38%

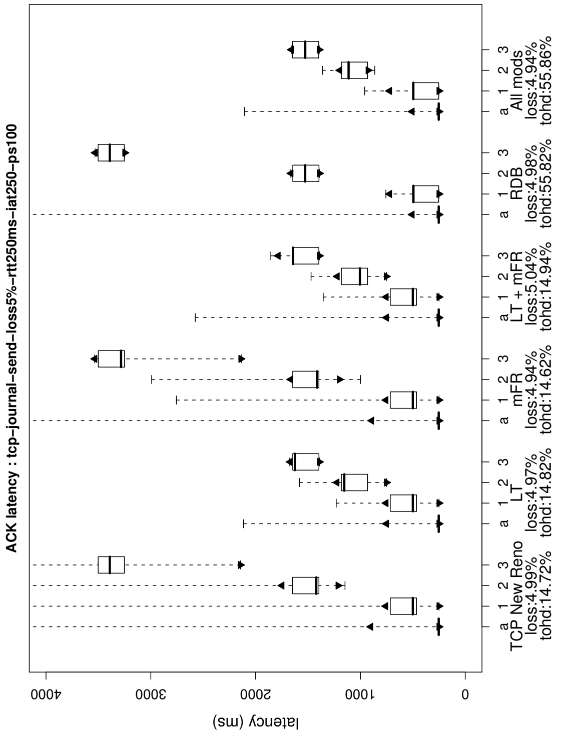

ACK latency : tcp–journal–send–loss5%–rtt200ms–iat25–ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.95%
tohd:5.30%
a 1 2 3
LT
loss:4.98%
tohd:5.33%
a 1 2 3
mFR
loss:4.97%
tohd:5.40%
a 1 2 3
LT + mFR
loss:4.98%
tohd:5.39%
a 1 2 3
RDB
loss:5.01%
tohd:89.02%
a 1 2 3
All mods
loss:5.00%
tohd:89.02%

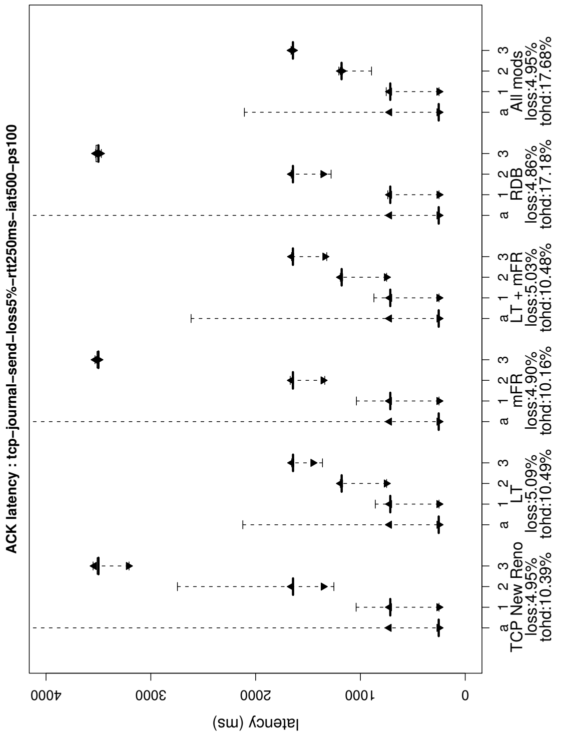

ACK latency : tcp-journal-send-loss5%-rtt250ms-iat100-ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.92%
tohd:5.24%
a 1 2 3
LT
loss:5.01%
tohd:5.40%
a 1 2 3
mFR
loss:5.01%
tohd:5.43%
a 1 2 3
LT + mFR
loss:4.93%
tohd:5.42%
a 1 2 3
RDB
loss:4.94%
tohd:67.82%
a 1 2 3
All mods
loss:5.03%
tohd:67.84%

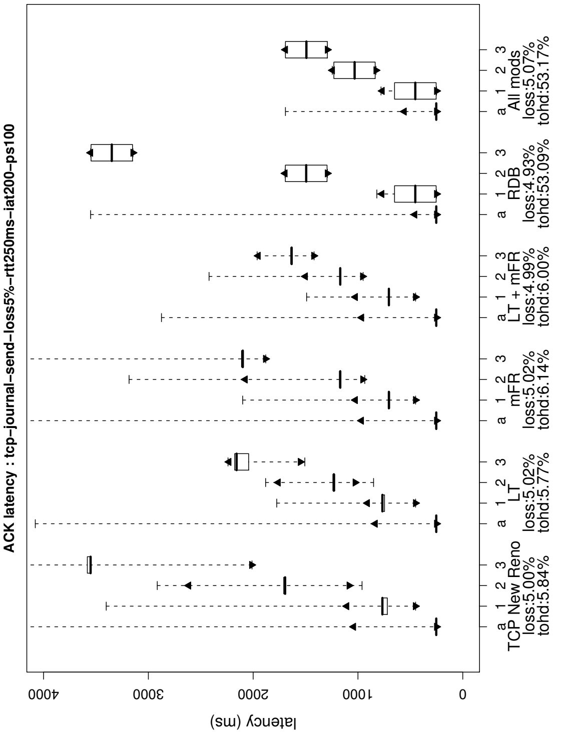

ACK latency : tcp-journal-send-loss5%-rtt200ms-iat500-ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:5.10%
tohd:10.53%
a 1 2 3
LT
loss:4.99%
tohd:10.36%
a 1 2 3
mFR
loss:4.96%
tohd:10.23%
a 1 2 3
LT + mFR
loss:4.86%
tohd:9.97%
a 1 2 3
RDB
loss:4.88%
tohd:18.14%
a 1 2 3
All mods
loss:4.99%
tohd:17.45%

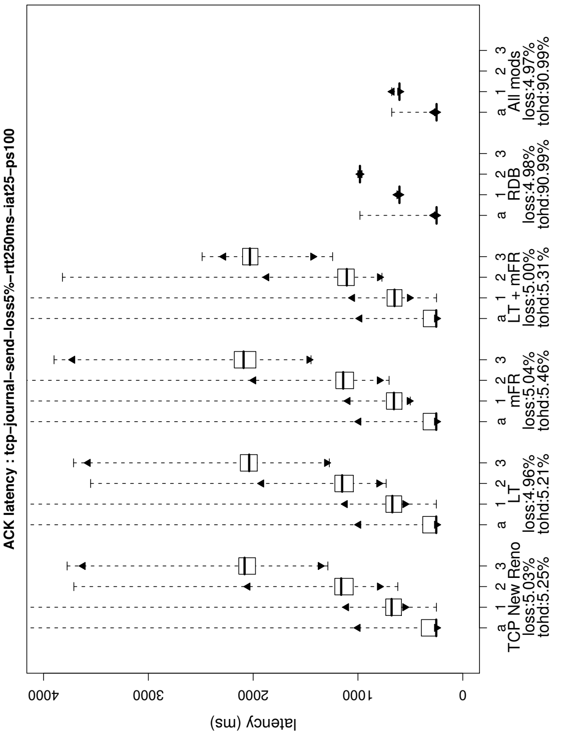

ACK latency : tcp-journal-send-loss5%-rtt250ms-iat150-ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.99%
tohd:5.62%
a 1 2 3
LT
loss:5.03%
tohd:5.63%
a 1 2 3
mFR
loss:4.91%
tohd:5.56%
a 1 2 3
LT + mFR
loss:5.04%
tohd:5.74%
a 1 2 3
RDB
loss:5.07%
tohd:52.69%
a 1 2 3
All mods
loss:5.01%
tohd:52.66%

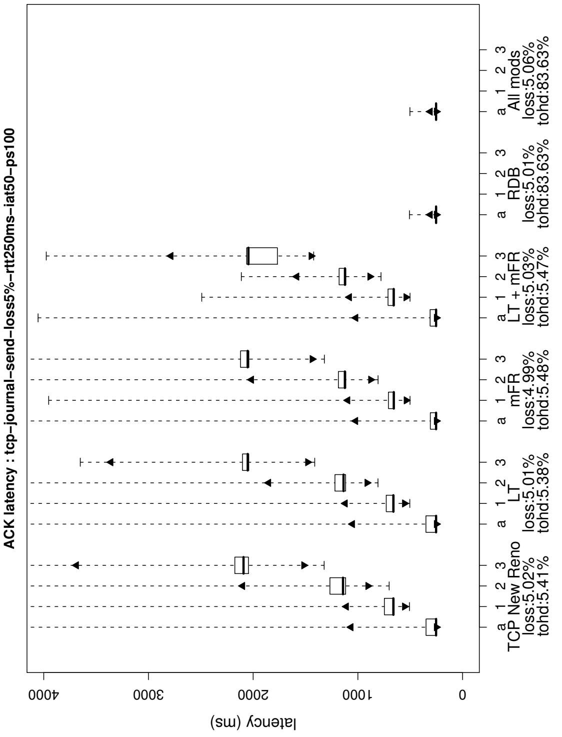

ACK latency : tcp-journal-send-loss5%-rtt200ms-iat50-ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.98%
tohd:5.37%
a 1 2 3
LT
loss:5.05%
tohd:5.45%
a 1 2 3
mFR
loss:5.05%
tohd:5.49%
a 1 2 3
LT + mFR
loss:4.99%
tohd:5.40%
a 1 2 3
RDB
loss:5.00%
tohd:80.42%
a 1 2 3
All mods
loss:4.95%
tohd:80.42%

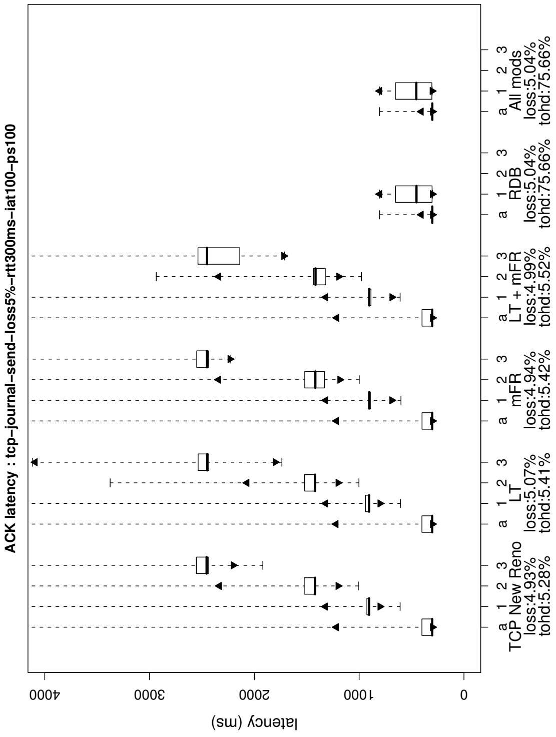

ACK latency : tcp−journal−send−loss5%−rtt250ms−iat250−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.99%
tohd:14.72%
a 1 2 3
LT
loss:4.97%
tohd:14.82%
a 1 2 3
mFR
loss:4.94%
tohd:14.62%
a 1 2 3
LT + mFR
loss:5.04%
tohd:14.94%
a 1 2 3
RDB
loss:4.98%
tohd:55.82%
a 1 2 3
All mods
loss:4.94%
tohd:55.86%

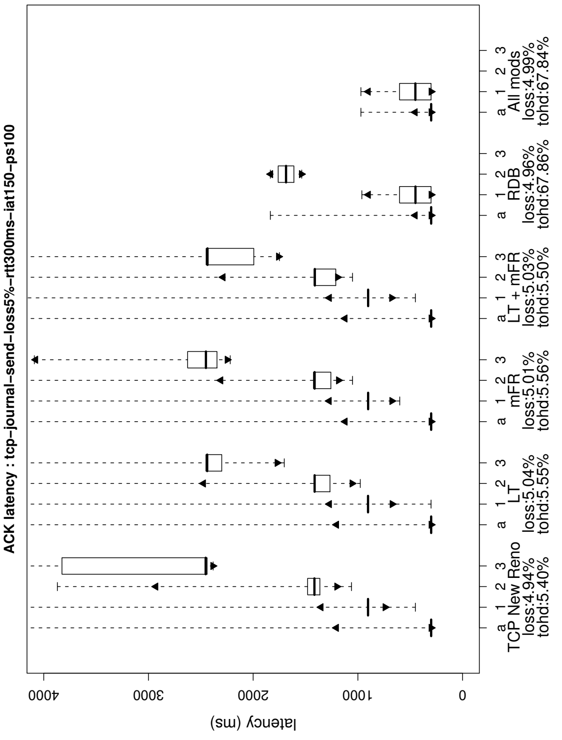

ACK latency : tcp−journal−send−loss5%−rtt250ms−iat500−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.95%
tohd:10.39%
a 1 2 3
LT
loss:5.09%
tohd:10.49%
a 1 2 3
mFR
loss:4.90%
tohd:10.16%
a 1 2 3
LT + mFR
loss:5.03%
tohd:10.48%
a 1 2 3
RDB
loss:4.86%
tohd:17.18%
a 1 2 3
All mods
loss:4.95%
tohd:17.68%

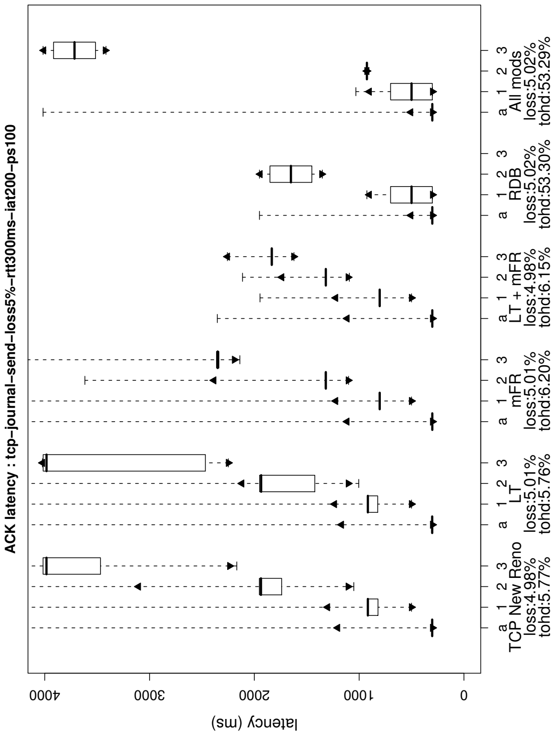

ACK latency : tcp−journal−send−loss5%−rtt250ms−iat200−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:5.00%
tohd:5.84%
a 1 2 3
LT
loss:5.02%
tohd:5.77%
a 1 2 3
mFR
loss:5.02%
tohd:6.14%
a 1 2 3
LT + mFR
loss:4.99%
tohd:6.00%
a 1 2 3
RDB
loss:4.93%
tohd:53.09%
a 1 2 3
All mods
loss:5.07%
tohd:53.17%

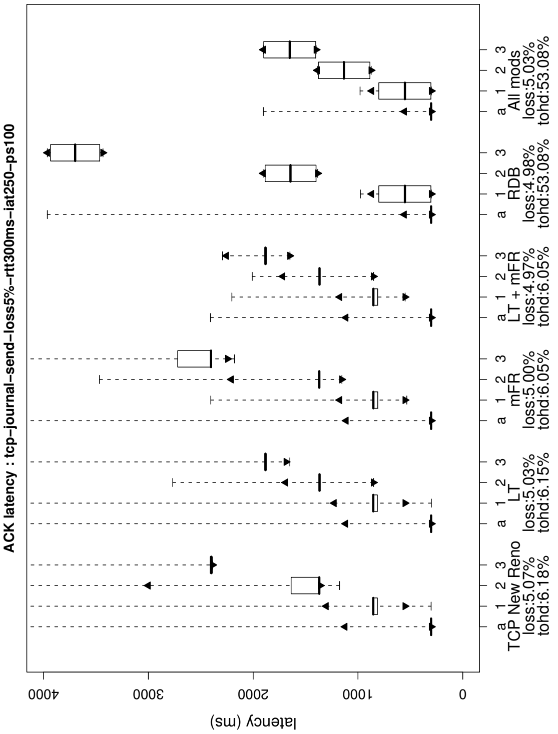

ACK latency : tcp−journal−send−loss5%−rtt250ms−iat25−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:5.03%
tohd:5.25%
a 1 2 3
LT
loss:4.96%
tohd:5.21%
a 1 2 3
mFR
loss:5.04%
tohd:5.46%
a 1 2 3
LT + mFR
loss:5.00%
tohd:5.31%
a 1 2 3
RDB
loss:4.98%
tohd:90.99%
a 1 2 3
All mods
loss:4.97%
tohd:90.99%

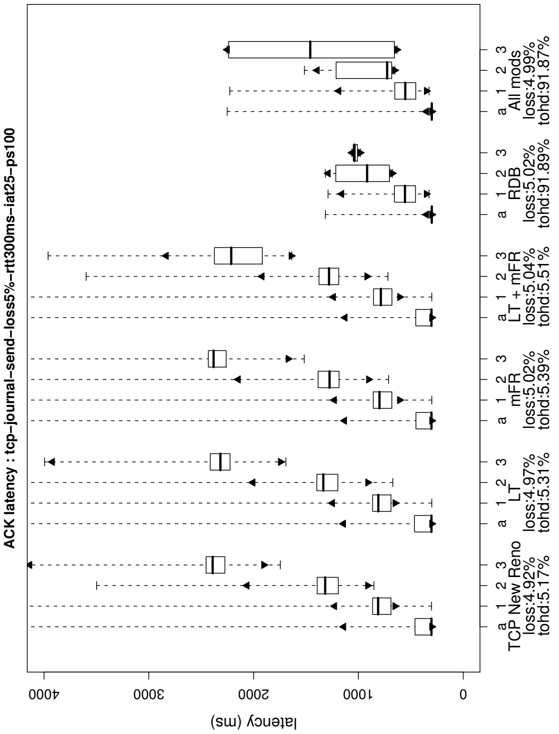

ACK latency : tcp−journal−send−loss5%−rtt250ms−lat50−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:5.02%
tohd:5.41%
a 1 2 3
LT
loss:5.01%
tohd:5.38%
a 1 2 3
mFR
loss:4.99%
tohd:5.48%
a 1 2 3
LT + mFR
loss:5.03%
tohd:5.47%
a 1 2 3
RDB
loss:5.01%
tohd:83.63%
a 1 2 3
All mods
loss:5.06%
tohd:83.63%

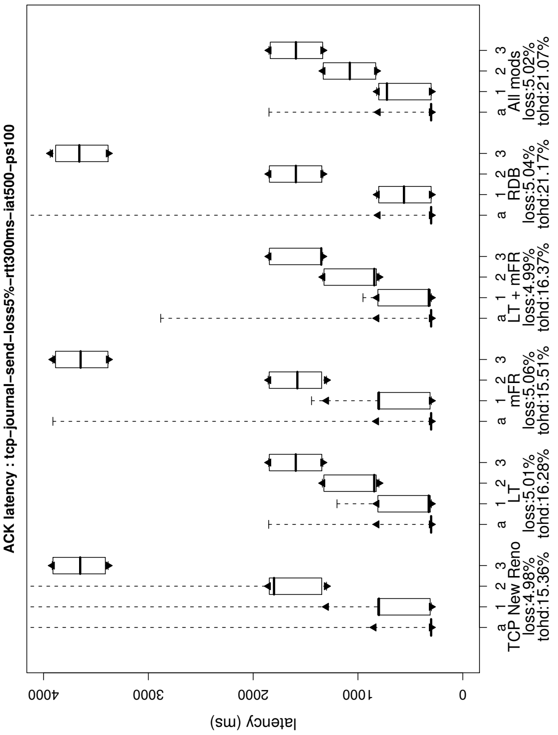

ACK latency : tcp−journal−send−loss5%−rtt300ms−lat100−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.93%
tohd:5.28%
a 1 2 3
LT
loss:5.07%
tohd:5.41%
a 1 2 3
mFR
loss:4.94%
tohd:5.42%
a 1 2 3
LT + mFR
loss:4.99%
tohd:5.52%
a 1 2 3
RDB
loss:5.04%
tohd:75.66%
a 1 2 3
All mods
loss:5.04%
tohd:75.66%

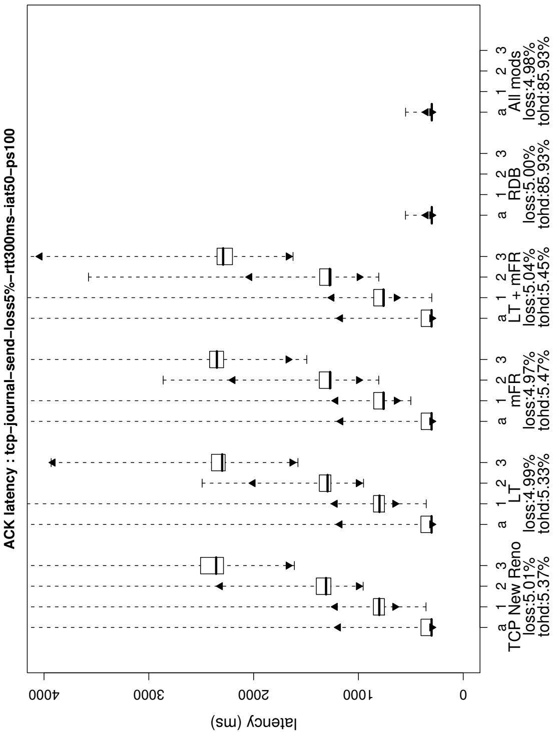

ACK latency : tcp−journal−send−loss5%−rtt300ms−lat150−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.94%
tohd:5.40%
a 1 2 3
LT
loss:5.04%
tohd:5.55%
a 1 2 3
mFR
loss:5.01%
tohd:5.56%
a 1 2 3
LT + mFR
loss:5.03%
tohd:5.50%
a 1 2 3
RDB
loss:4.96%
tohd:67.86%
a 1 2 3
All mods
loss:4.99%
tohd:67.84%

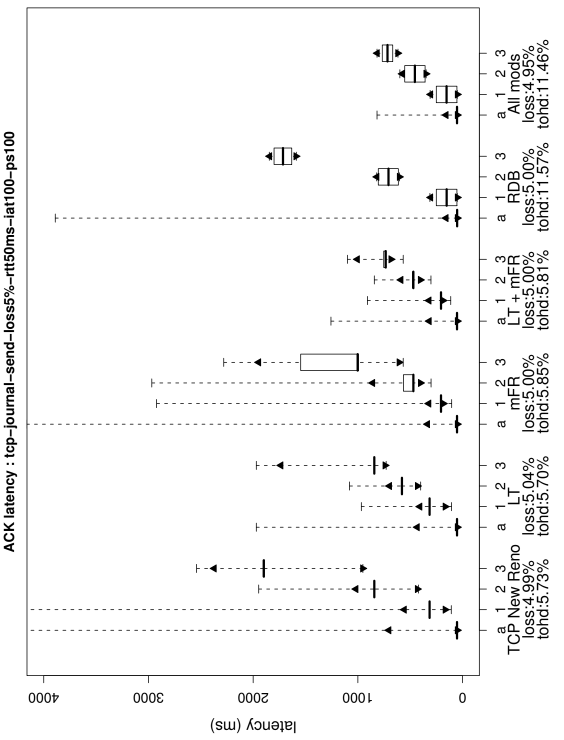

ACK latency : tcp−journal−send−loss5%−rtt300ms−lat200−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.98%
tohd:5.77%
a 1 2 3
LT
loss:5.01%
tohd:5.76%
a 1 2 3
mFR
loss:5.01%
tohd:6.20%
a 1 2 3
LT + mFR
loss:4.98%
tohd:6.15%
a 1 2 3
RDB
loss:5.02%
tohd:53.30%
a 1 2 3
All mods
loss:5.02%
tohd:53.29%

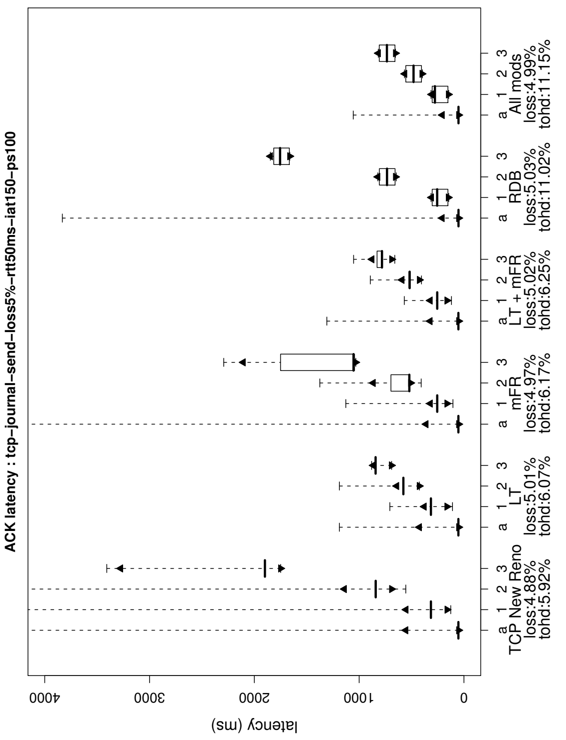

ACK latency : tcp−journal−send−loss5%−rtt300ms−iat250−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:5.07%
tohd:6.18%
a 1 2 3
LT
loss:5.03%
tohd:6.15%
a 1 2 3
mFR
loss:5.00%
tohd:6.05%
a 1 2 3
LT + mFR
loss:4.97%
tohd:6.05%
a 1 2 3
RDB
loss:4.98%
tohd:53.08%
a 1 2 3
All mods
loss:5.03%
tohd:53.08%

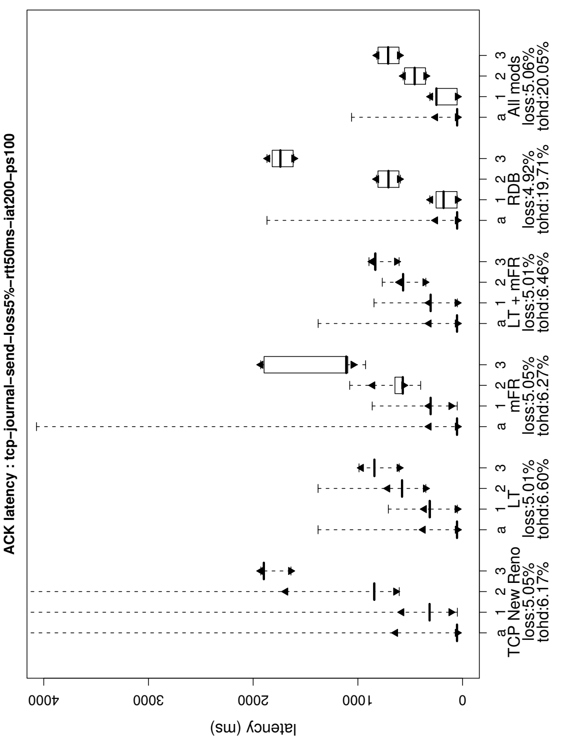

ACK latency : tcp−journal−send−loss5%−rtt300ms−iat25−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.92%
tohd:5.17%
a 1 2 3
LT
loss:4.97%
tohd:5.31%
a 1 2 3
mFR
loss:5.02%
tohd:5.39%
a 1 2 3
LT + mFR
loss:5.04%
tohd:5.51%
a 1 2 3
RDB
loss:5.02%
tohd:91.89%
a 1 2 3
All mods
loss:4.99%
tohd:91.87%

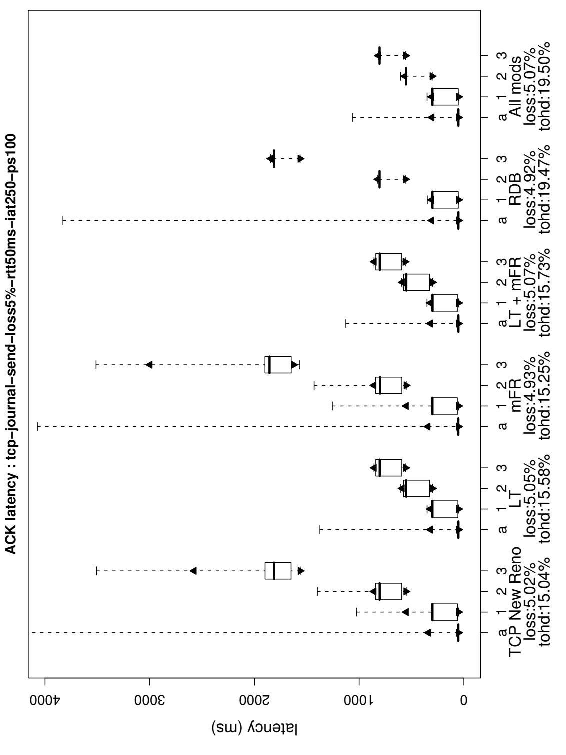

ACK latency : tcp−journal−send−loss5%−rtt300ms−iat500−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.98%
tohd:15.36%
a 1 2 3
LT
loss:5.01%
tohd:16.28%
a 1 2 3
mFR
loss:5.06%
tohd:15.51%
a 1 2 3
LT + mFR
loss:4.99%
tohd:16.37%
a 1 2 3
RDB
loss:5.04%
tohd:21.17%
a 1 2 3
All mods
loss:5.02%
tohd:21.07%

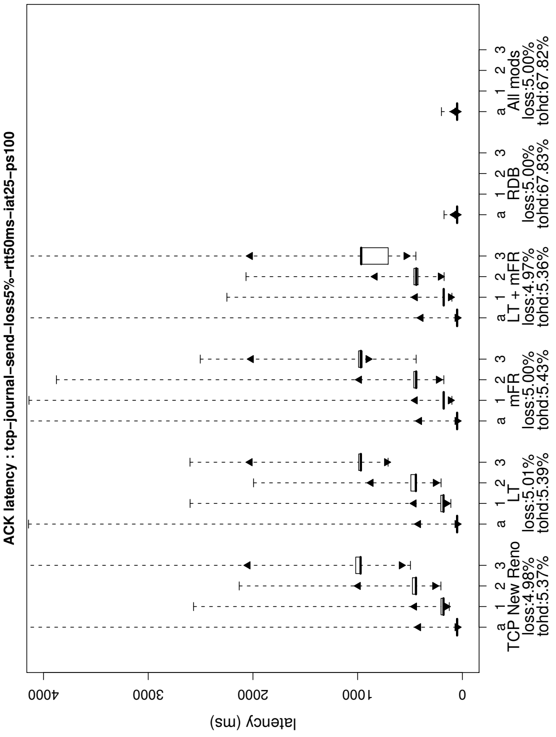

ACK latency : tcp−journal−send−loss5%−rtt300ms−iat50−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:5.01%
tohd:5.37%
a 1 2 3
LT
loss:4.99%
tohd:5.33%
a 1 2 3
mFR
loss:4.97%
tohd:5.47%
a 1 2 3
LT + mFR
loss:5.04%
tohd:5.45%
a 1 2 3
RDB
loss:5.00%
tohd:85.93%
a 1 2 3
All mods
loss:4.98%
tohd:85.93%

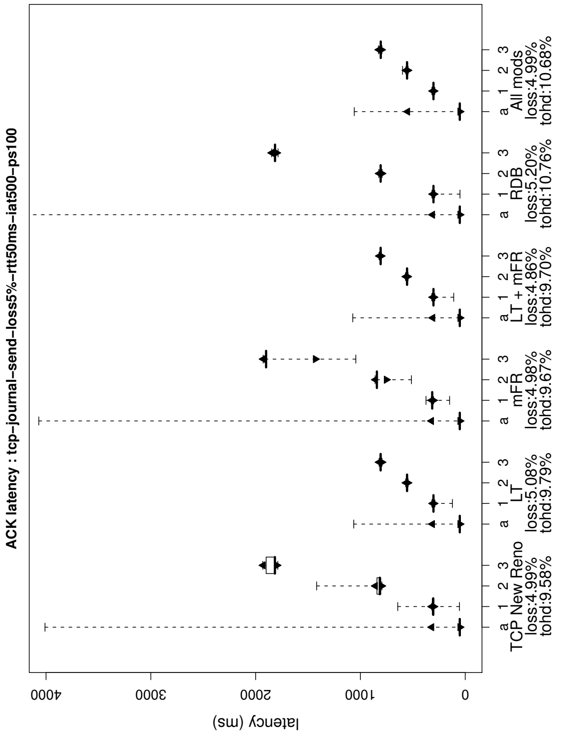

ACK latency : tcp–journal–send–loss5%–rtt50ms–iat100–ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.99%
tohd:5.73%
a 1 2 3
LT
loss:5.04%
tohd:5.70%
a 1 2 3
mFR
loss:5.00%
tohd:5.85%
a 1 2 3
LT + mFR
loss:5.00%
tohd:5.81%
a 1 2 3
RDB
loss:5.00%
tohd:11.57%
a 1 2 3
All mods
loss:4.95%
tohd:11.46%

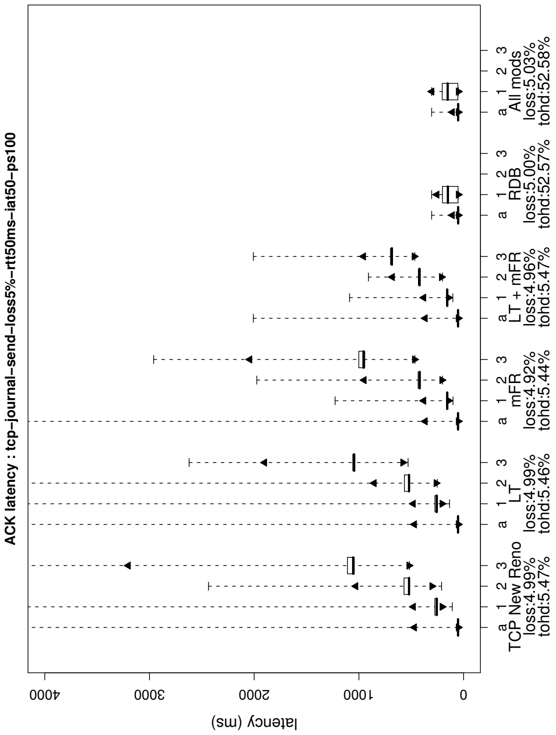

ACK latency : tcp–journal–send–loss5%–rtt50ms–iat150–ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.88%
tohd:5.92%
a 1 2 3
LT
loss:5.01%
tohd:6.07%
a 1 2 3
mFR
loss:4.97%
tohd:6.17%
a 1 2 3
LT + mFR
loss:5.02%
tohd:6.25%
a 1 2 3
RDB
loss:5.03%
tohd:11.02%
a 1 2 3
All mods
loss:4.99%
tohd:11.15%


ACK latency : tcp–journal–send–loss5%–rtt50ms–iat200–ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:5.05%
tohd:6.17%
a 1 2 3
LT
loss:5.01%
tohd:6.60%
a 1 2 3
mFR
loss:5.05%
tohd:6.27%
a 1 2 3
LT + mFR
loss:5.01%
tohd:6.46%
a 1 2 3
RDB
loss:4.92%
tohd:19.71%
a 1 2 3
All mods
loss:5.06%
tohd:20.05%

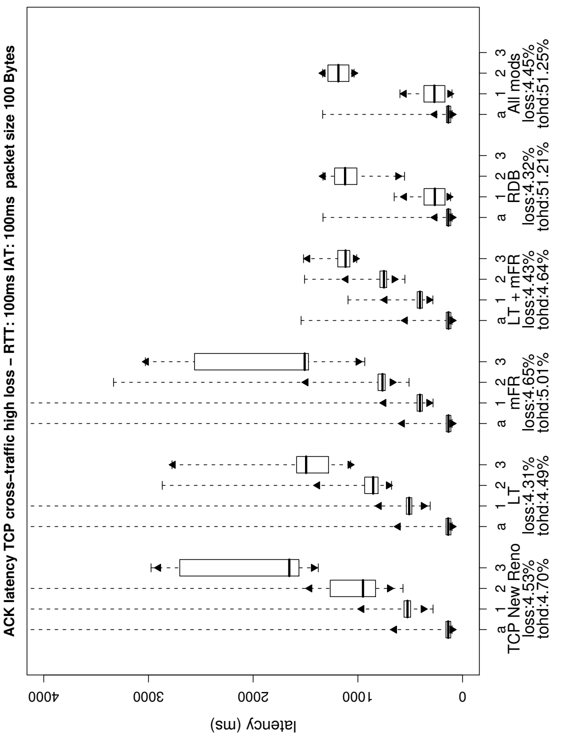

ACK latency : tcp–journal–send–loss5%–rtt50ms–iat250–ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:5.02%
tohd:15.04%
a 1 2 3
LT
loss:5.05%
tohd:15.58%
a 1 2 3
mFR
loss:4.93%
tohd:15.25%
a 1 2 3
LT + mFR
loss:5.07%
tohd:15.73%
a 1 2 3
RDB
loss:4.92%
tohd:19.47%
a 1 2 3
All mods
loss:5.07%
tohd:19.50%

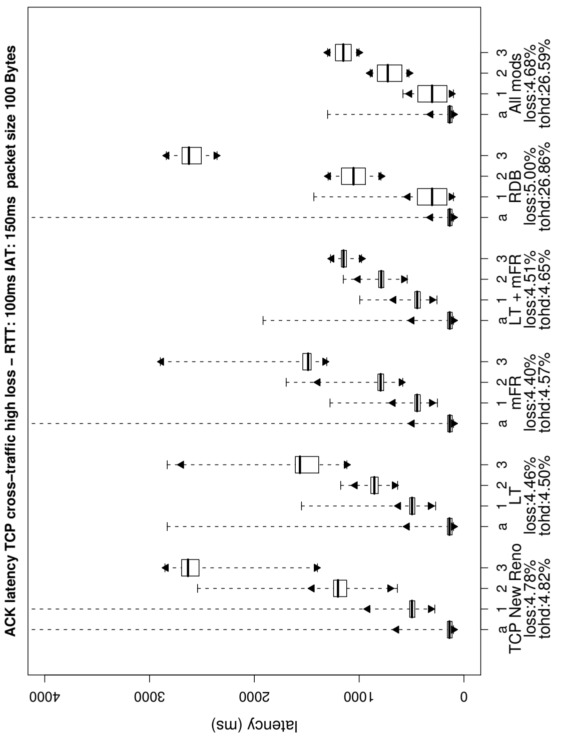

ACK latency : tcp−journal−send−loss5%−rtt50ms−iat25−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.98%
tohd:5.37%
a 1 2 3
LT
loss:5.01%
tohd:5.39%
a 1 2 3
mFR
loss:5.00%
tohd:5.43%
a 1 2 3
LT + mFR
loss:4.97%
tohd:5.36%
a 1 2 3
RDB
loss:5.00%
tohd:67.83%
a 1 2 3
All mods
loss:5.00%
tohd:67.82%

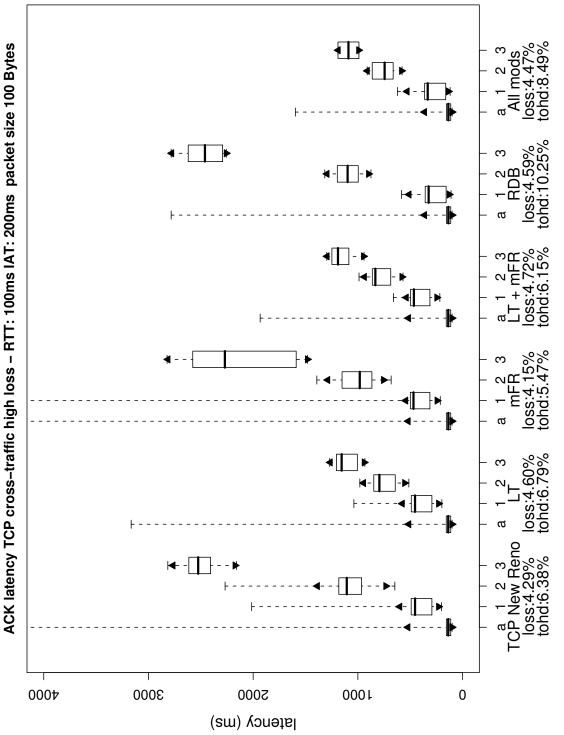

ACK latency : tcp−journal−send−loss5%−rtt50ms−iat500−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.99%
tohd:9.58%
a 1 2 3
LT
loss:5.08%
tohd:9.79%
a 1 2 3
mFR
loss:4.98%
tohd:9.67%
a 1 2 3
LT + mFR
loss:4.86%
tohd:9.70%
a 1 2 3
RDB
loss:5.20%
tohd:10.76%
a 1 2 3
All mods
loss:4.99%
tohd:10.68%

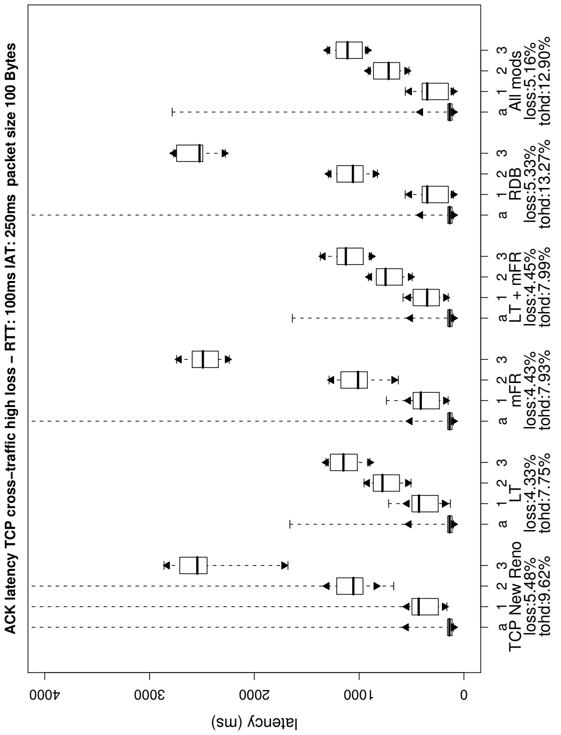

ACK latency : tcp−journal−send−loss5%−rtt50ms−iat50−ps100
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.99%
tohd:5.47%
a 1 2 3
LT
loss:4.99%
tohd:5.46%
a 1 2 3
mFR
loss:4.92%
tohd:5.44%
a 1 2 3
LT + mFR
loss:4.96%
tohd:5.47%
a 1 2 3
RDB
loss:5.00%
tohd:52.57%
a 1 2 3
All mods
loss:5.03%
tohd:52.58%

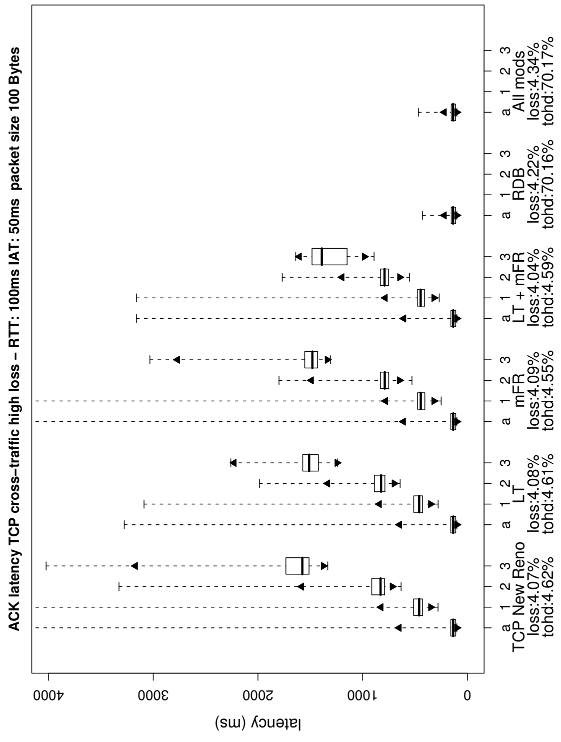
C.2 Cross-traffic loss - high loss rate

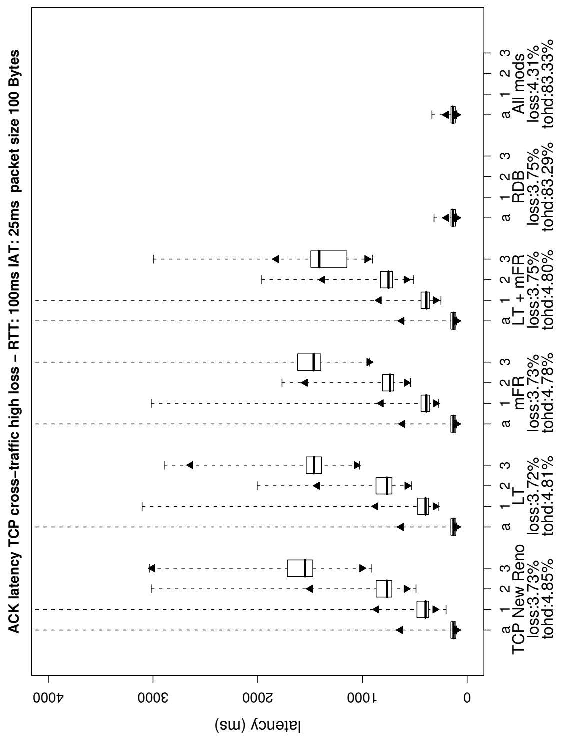


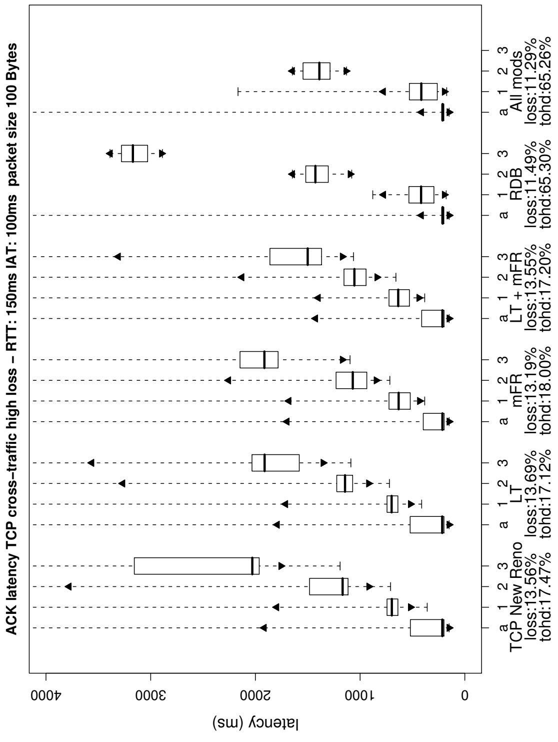


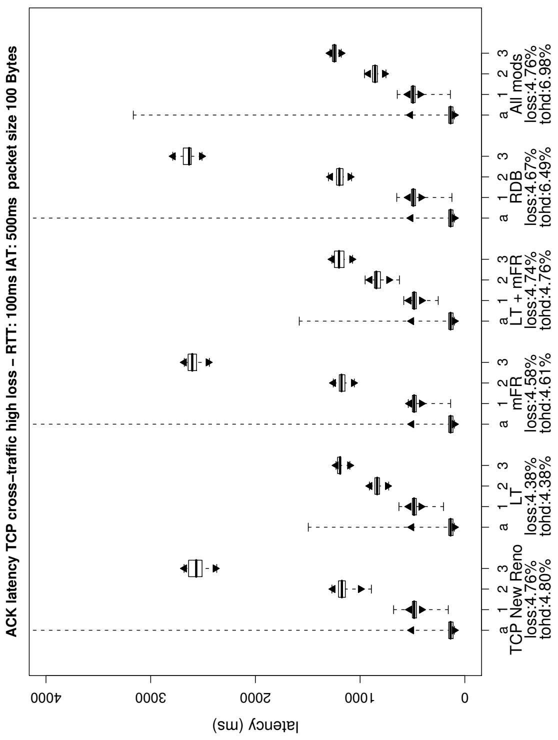


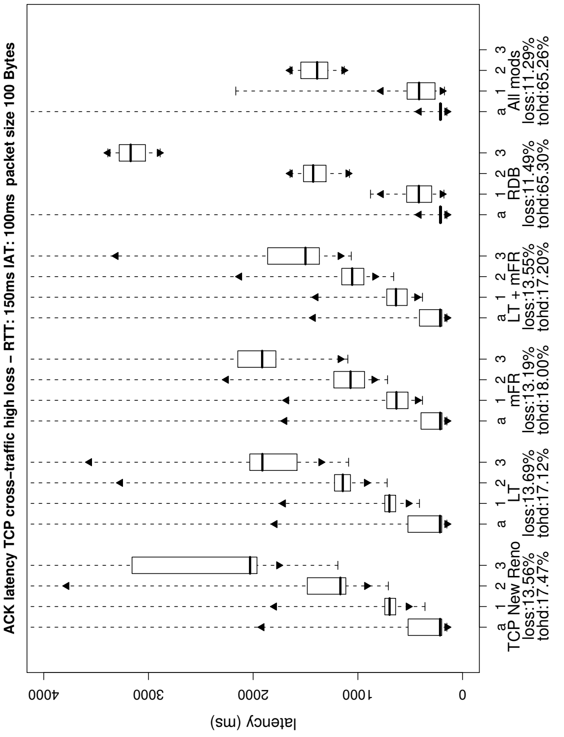


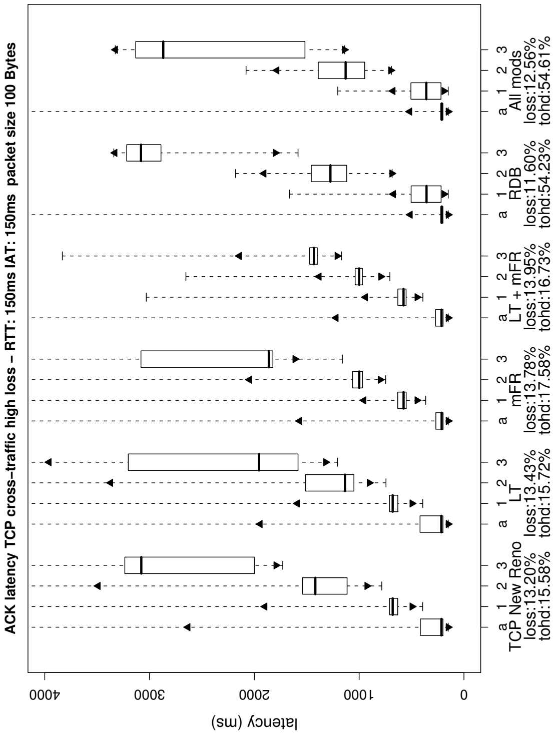

ACK latency TCP cross−traffic high loss − RTT: 100ms IAT: 50ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.07%
tohd:4.62%
a 1 2 3
LT
loss:4.08%
tohd:4.61%
a 1 2 3
mFR
loss:4.09%
tohd:4.55%
a 1 2 3
LT + mFR
loss:4.04%
tohd:4.59%
a 1 2 3
RDB
loss:4.22%
tohd:70.16%
a 1 2 3
All mods
loss:4.34%
tohd:70.17%

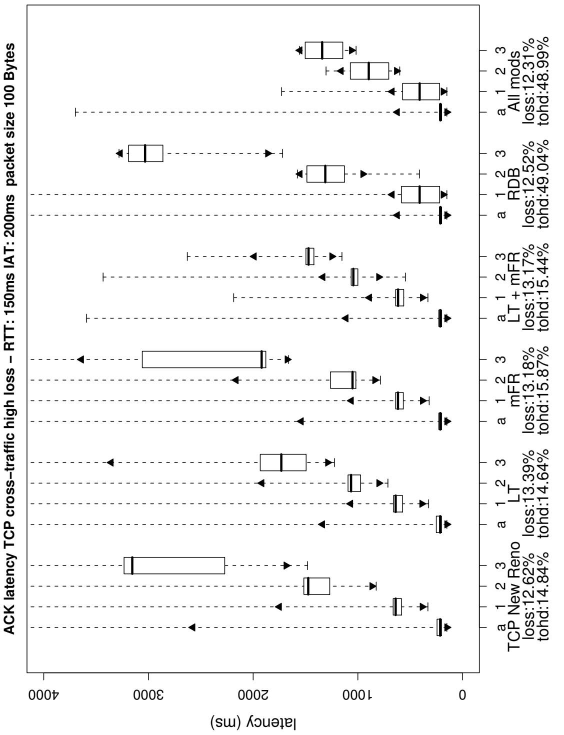

ACK latency TCP cross−traffic high loss − RTT: 100ms IAT: 25ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:3.73%
tohd:4.85%
a 1 2 3
LT
loss:3.72%
tohd:4.81%
a 1 2 3
mFR
loss:3.73%
tohd:4.78%
a 1 2 3
LT + mFR
loss:3.75%
tohd:4.80%
a 1 2 3
RDB
loss:3.75%
tohd:83.29%
a 1 2 3
All mods
loss:4.31%
tohd:83.33%

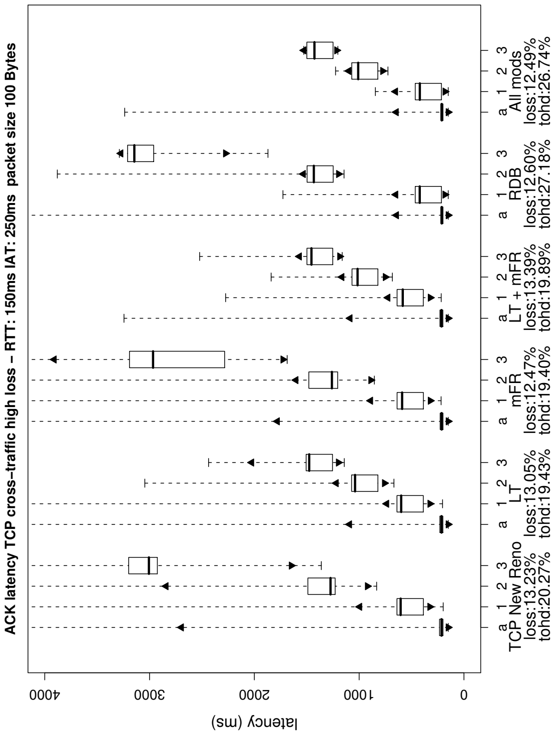

ACK latency TCP cross−traffic high loss − RTT: 150ms IAT: 100ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:13.56%
tohd:17.47%
a 1 2 3
LT
loss:13.69%
tohd:17.12%
a 1 2 3
mFR
loss:13.19%
tohd:18.00%
a 1 2 3
LT + mFR
loss:13.55%
tohd:17.20%
a 1 2 3
RDB
loss:11.49%
tohd:65.30%
a 1 2 3
All mods
loss:11.29%
tohd:65.26%

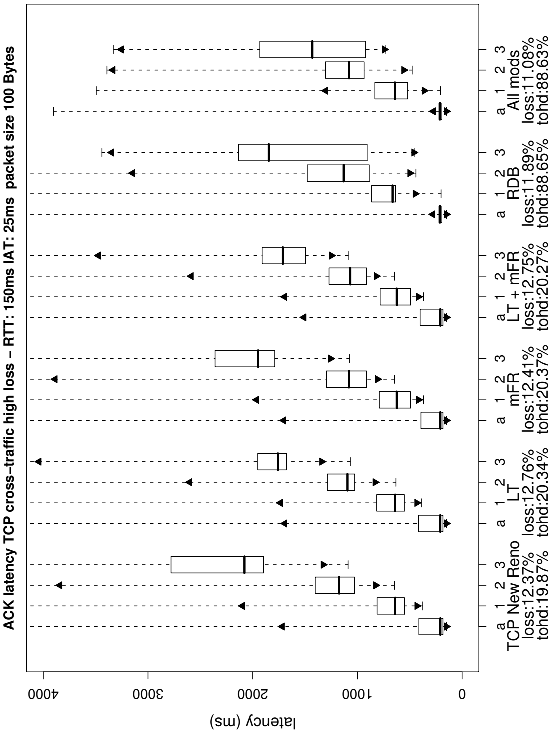

ACK latency TCP cross−traffic high loss − RTT: 100ms IAT: 500ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:4.76%
tohd:4.80%
a 1 2 3
LT
loss:4.38%
tohd:4.38%
a 1 2 3
mFR
loss:4.58%
tohd:4.61%
a 1 2 3
LT + mFR
loss:4.74%
tohd:4.76%
a 1 2 3
RDB
loss:4.67%
tohd:6.49%
a 1 2 3
All mods
loss:4.76%
tohd:6.98%

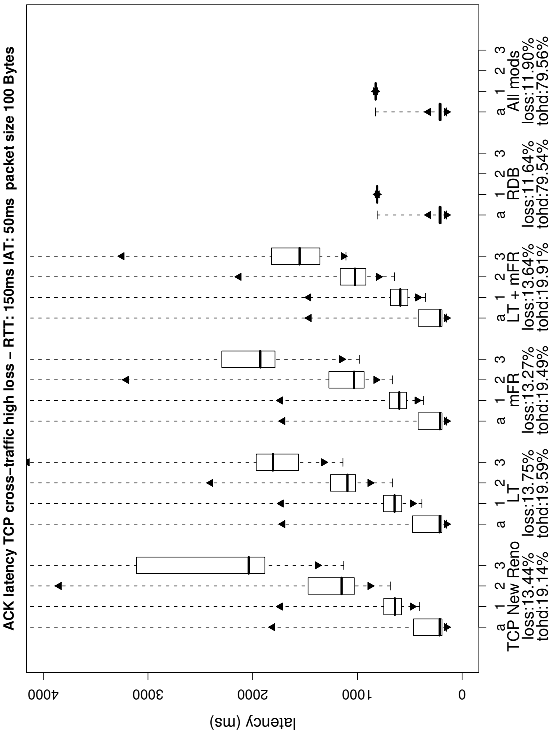

ACK latency TCP cross-traffic high loss – RTT: 150ms IAT: 100ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:13.56%
tohd:17.47%
a 1 2 3
LT
loss:13.69%
tohd:17.12%
a 1 2 3
mFR
loss:13.19%
tohd:18.00%
a 1 2 3
LT + mFR
loss:13.55%
tohd:17.20%
a 1 2 3
RDB
loss:11.49%
tohd:65.30%
a 1 2 3
All mods
loss:11.29%
tohd:65.26%

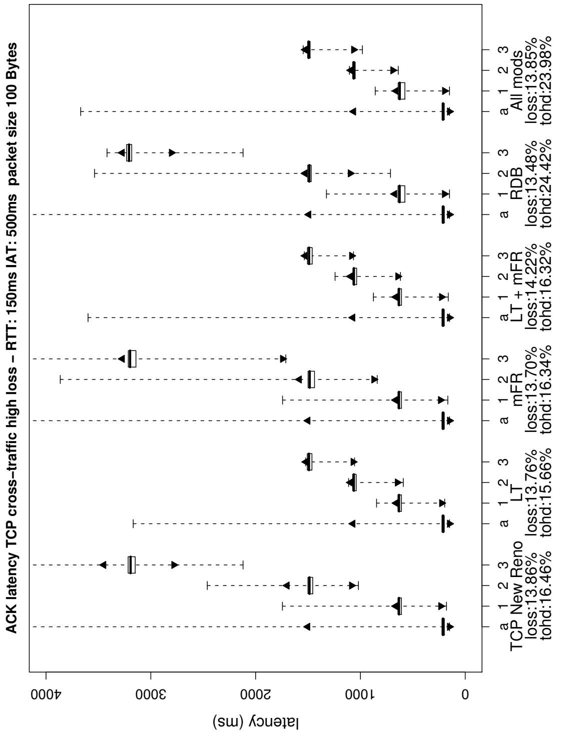

ACK latency TCP cross-traffic high loss – RTT: 150ms IAT: 150ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:13.20%
tohd:15.58%
a 1 2 3
LT
loss:13.43%
tohd:15.72%
a 1 2 3
mFR
loss:13.78%
tohd:17.58%
a 1 2 3
LT + mFR
loss:13.95%
tohd:16.73%
a 1 2 3
RDB
loss:11.60%
tohd:54.23%
a 1 2 3
All mods
loss:12.56%
tohd:54.61%

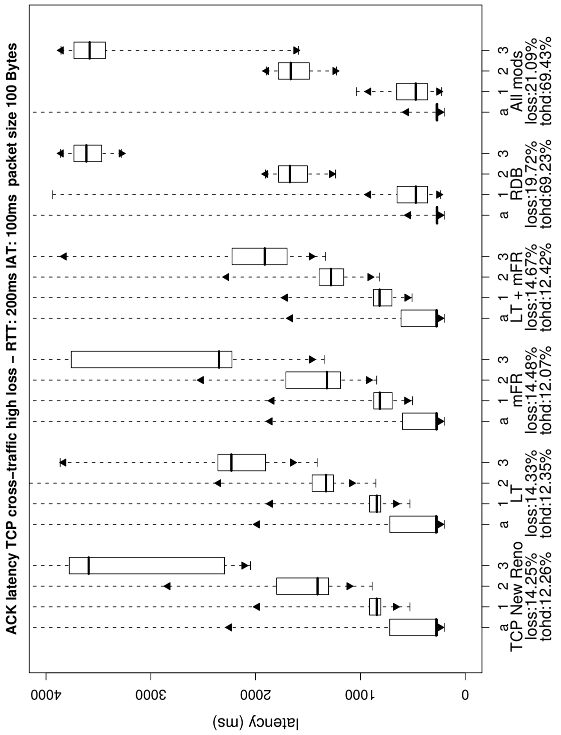

ACK latency TCP cross-traffic high loss – RTT: 150ms IAT: 200ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:12.62%
tohd:14.84%
a 1 2 3
LT
loss:13.39%
tohd:14.64%
a 1 2 3
mFR
loss:13.18%
tohd:15.87%
a 1 2 3
LT + mFR
loss:13.17%
tohd:15.44%
a 1 2 3
RDB
loss:12.52%
tohd:49.04%
a 1 2 3
All mods
loss:12.31%
tohd:48.99%

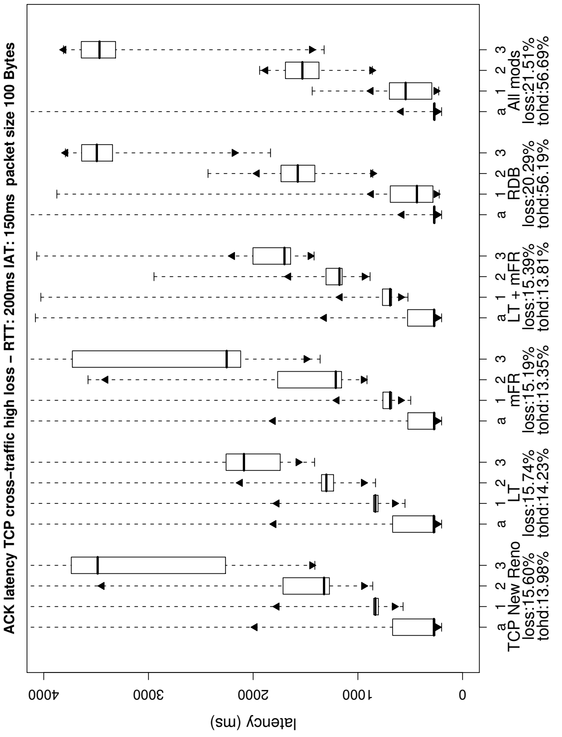

ACK latency TCP cross-traffic high loss – RTT: 150ms IAT: 250ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:13.23%
tohd:20.27%
a 1 2 3
LT
loss:13.05%
tohd:19.43%
a 1 2 3
mFR
loss:12.47%
tohd:19.40%
a 1 2 3
LT + mFR
loss:13.39%
tohd:19.89%
a 1 2 3
RDB
loss:12.60%
tohd:27.18%
a 1 2 3
All mods
loss:12.49%
tohd:26.74%

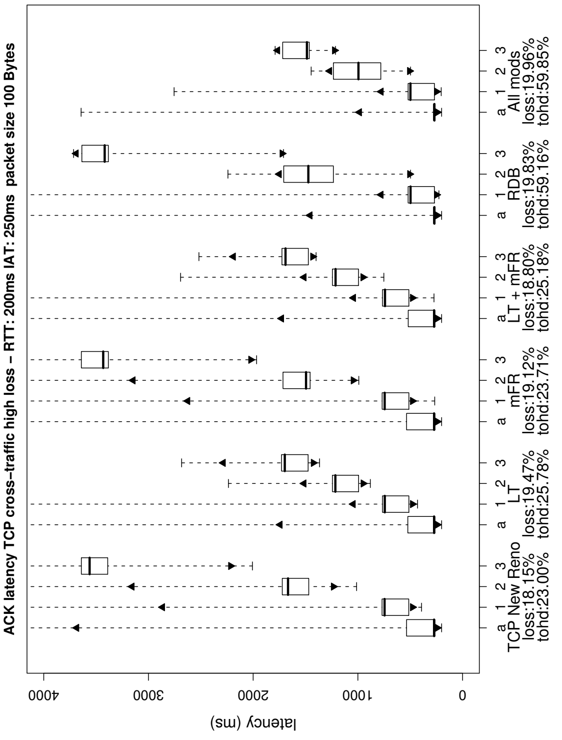

ACK latency TCP cross−traffic high loss − RTT: 150ms IAT: 25ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:12.37%
tohd:19.87%
a 1 2 3
LT
loss:12.76%
tohd:20.34%
a 1 2 3
mFR
loss:12.41%
tohd:20.37%
a 1 2 3
LT + mFR
loss:12.75%
tohd:20.27%
a 1 2 3
RDB
loss:11.89%
tohd:88.65%
a 1 2 3
All mods
loss:11.08%
tohd:88.63%

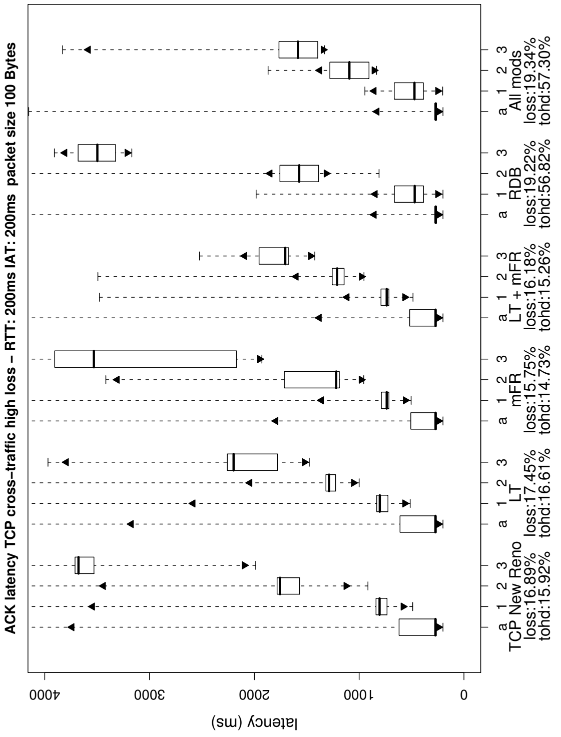

ACK latency TCP cross−traffic high loss − RTT: 150ms IAT: 50ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:13.44%
tohd:19.14%
a 1 2 3
LT
loss:13.75%
tohd:19.59%
a 1 2 3
mFR
loss:13.27%
tohd:19.49%
a 1 2 3
LT + mFR
loss:13.64%
tohd:19.91%
a 1 2 3
RDB
loss:11.64%
tohd:79.54%
a 1 2 3
All mods
loss:11.90%
tohd:79.56%

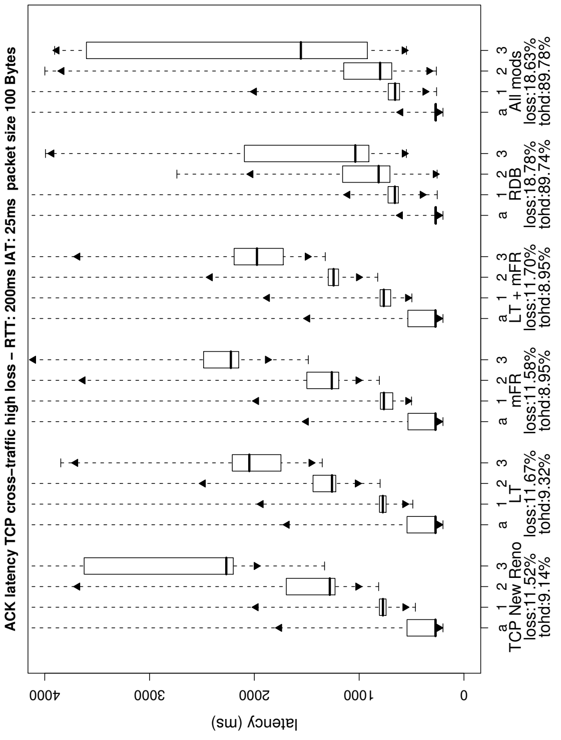

ACK latency TCP cross−traffic high loss − RTT: 150ms IAT: 500ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:13.86%
tohd:16.46%
a 1 2 3
LT
loss:13.76%
tohd:15.66%
a 1 2 3
mFR
loss:13.70%
tohd:16.34%
a 1 2 3
LT + mFR
loss:14.22%
tohd:16.32%
a 1 2 3
RDB
loss:13.48%
tohd:24.42%
a 1 2 3
All mods
loss:13.85%
tohd:23.98%

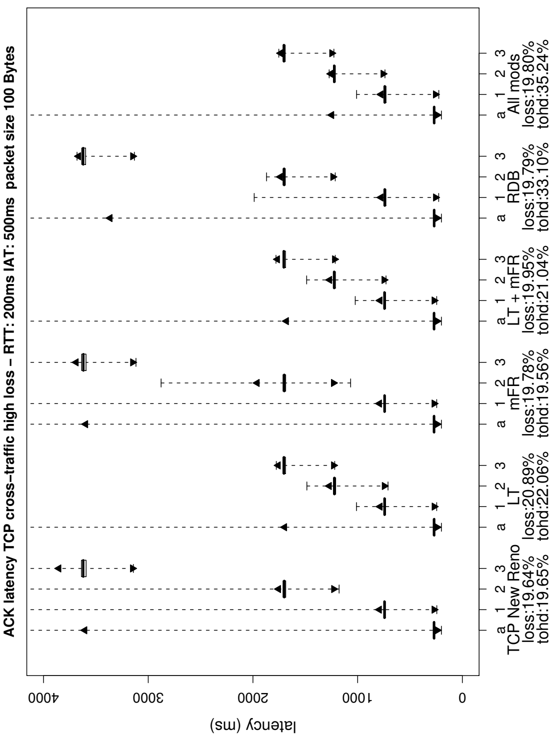

ACK latency TCP cross−traffic high loss − RTT: 200ms IAT: 100ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:14.25%
tohd:12.26%
a 1 2 3
LT
loss:14.33%
tohd:12.35%
a 1 2 3
mFR
loss:14.48%
tohd:12.07%
a 1 2 3
LT + mFR
loss:14.67%
tohd:12.42%
a 1 2 3
RDB
loss:19.72%
tohd:69.23%
a 1 2 3
All mods
loss:21.09%
tohd:69.43%

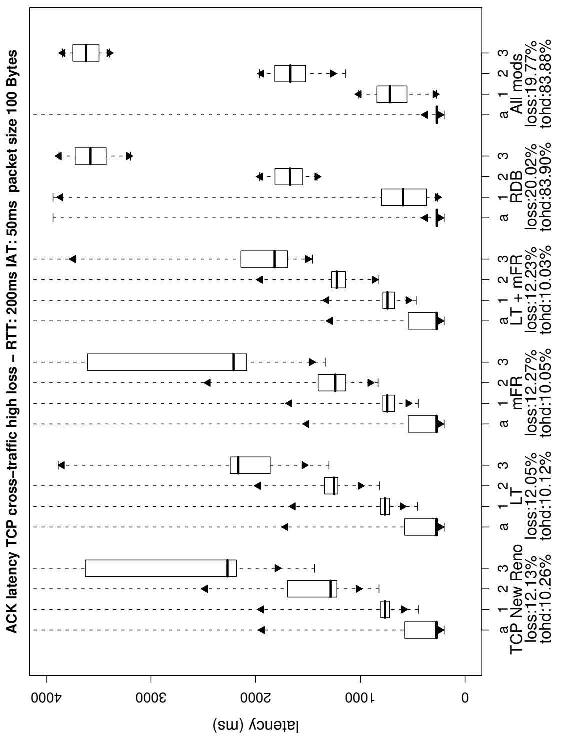

ACK latency TCP cross-traffic high loss – RTT: 200ms IAT: 150ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:15.60%
tohd:13.98%
a 1 2 3
LT
loss:15.74%
tohd:14.23%
a 1 2 3
mFR
loss:15.19%
tohd:13.35%
a 1 2 3
LT + mFR
loss:15.39%
tohd:13.81%
a 1 2 3
RDB
loss:20.29%
tohd:56.19%
a 1 2 3
All mods
loss:21.51%
tohd:56.69%

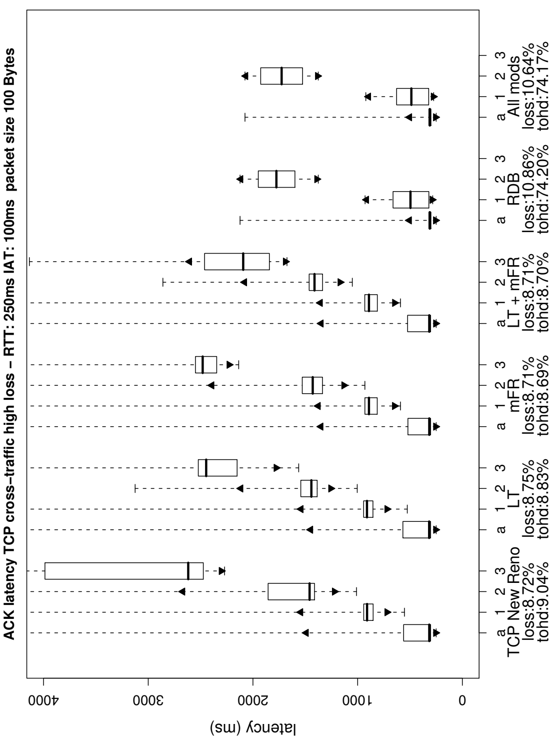

ACK latency TCP cross-traffic high loss – RTT: 200ms IAT: 250ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:18.15%
tohd:23.00%
a 1 2 3
LT
loss:19.47%
tohd:25.78%
a 1 2 3
mFR
loss:19.12%
tohd:23.71%
a 1 2 3
LT + mFR
loss:18.80%
tohd:25.18%
a 1 2 3
RDB
loss:19.83%
tohd:59.16%
a 1 2 3
All mods
loss:19.96%
tohd:59.85%

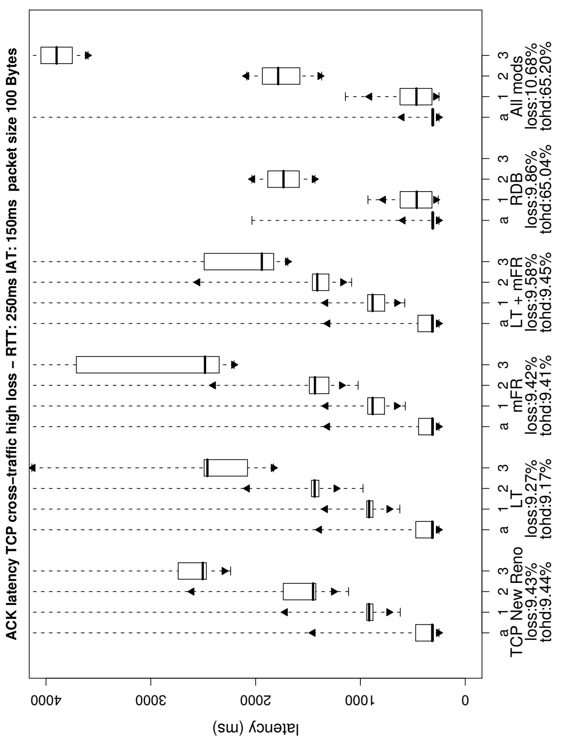

ACK latency TCP cross-traffic high loss – RTT: 200ms IAT: 200ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:16.89%
tohd:15.92%
a 1 2 3
LT
loss:17.45%
tohd:16.61%
a 1 2 3
mFR
loss:15.75%
tohd:14.73%
a 1 2 3
LT + mFR
loss:16.18%
tohd:15.26%
a 1 2 3
RDB
loss:19.22%
tohd:56.82%
a 1 2 3
All mods
loss:19.34%
tohd:57.30%

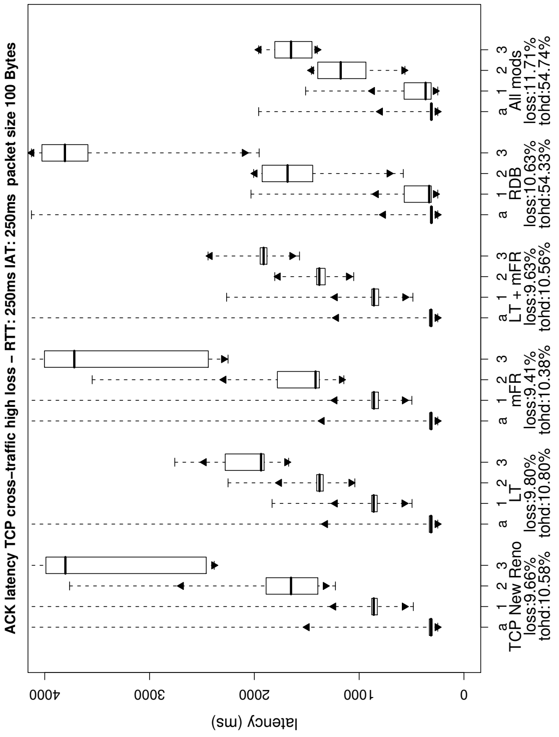

ACK latency TCP cross-traffic high loss – RTT: 200ms IAT: 25ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:11.52%
tohd:9.14%
a 1 2 3
LT
loss:11.67%
tohd:9.32%
a 1 2 3
mFR
loss:11.58%
tohd:8.95%
a 1 2 3
LT + mFR
loss:11.70%
tohd:8.95%
a 1 2 3
RDB
loss:18.78%
tohd:89.74%
a 1 2 3
All mods
loss:18.63%
tohd:89.78%

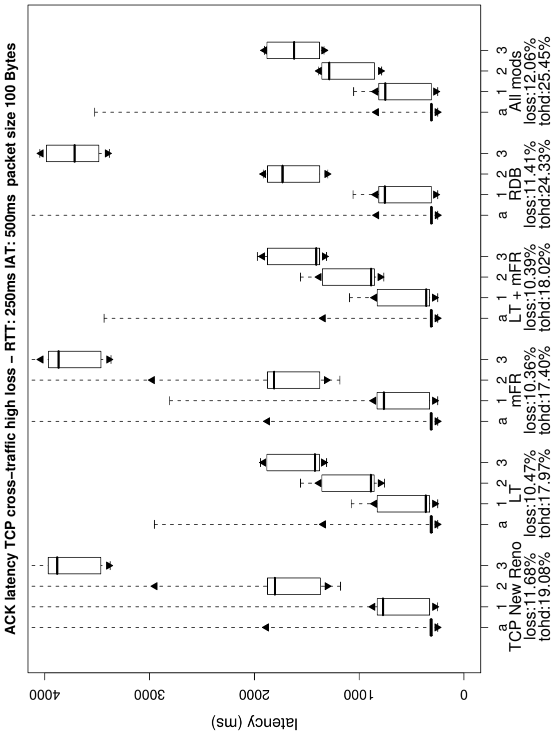

ACK latency TCP cross−traffic high loss − RTT: 200ms IAT: 500ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:19.64%
tohd:19.65%
a 1 2 3
LT
loss:20.89%
tohd:22.06%
a 1 2 3
mFR
loss:19.78%
tohd:19.56%
a 1 2 3
LT + mFR
loss:19.95%
tohd:21.04%
a 1 2 3
RDB
loss:19.79%
tohd:33.10%
a 1 2 3
All mods
loss:19.80%
tohd:35.24%

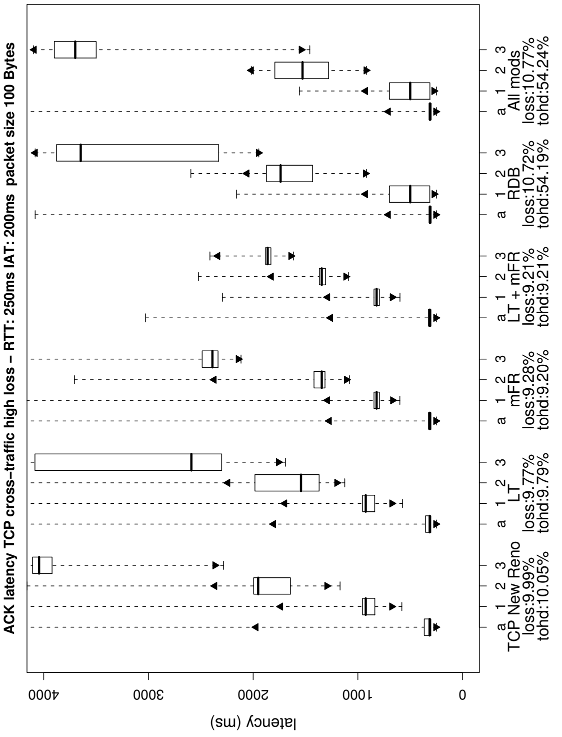

ACK latency TCP cross−traffic high loss − RTT: 200ms IAT: 50ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:12.13%
tohd:10.26%
a 1 2 3
LT
loss:12.05%
tohd:10.12%
a 1 2 3
mFR
loss:12.27%
tohd:10.05%
a 1 2 3
LT + mFR
loss:12.23%
tohd:10.03%
a 1 2 3
RDB
loss:20.02%
tohd:83.90%
a 1 2 3
All mods
loss:19.77%
tohd:83.88%

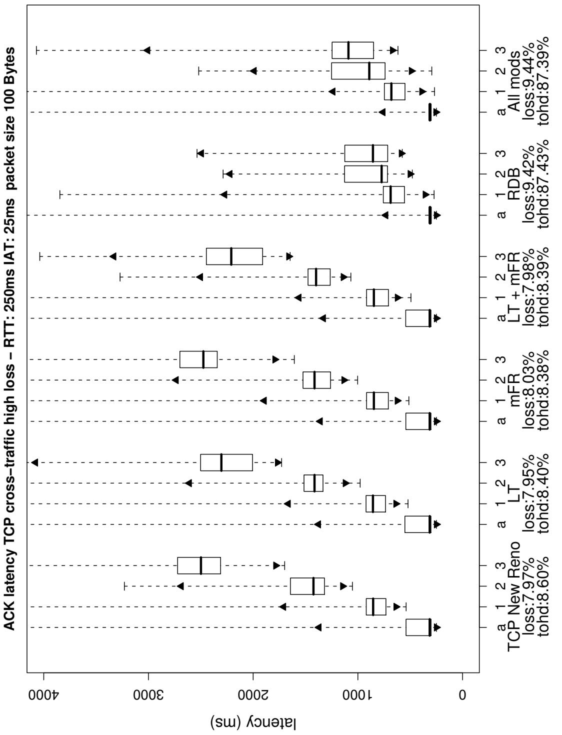

ACK latency TCP cross−traffic high loss − RTT: 250ms IAT: 100ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:8.72%
tohd:9.04%
a 1 2 3
LT
loss:8.75%
tohd:8.83%
a 1 2 3
mFR
loss:8.71%
tohd:8.69%
a 1 2 3
LT + mFR
loss:8.71%
tohd:8.70%
a 1 2 3
RDB
loss:10.86%
tohd:74.20%
a 1 2 3
All mods
loss:10.64%
tohd:74.17%

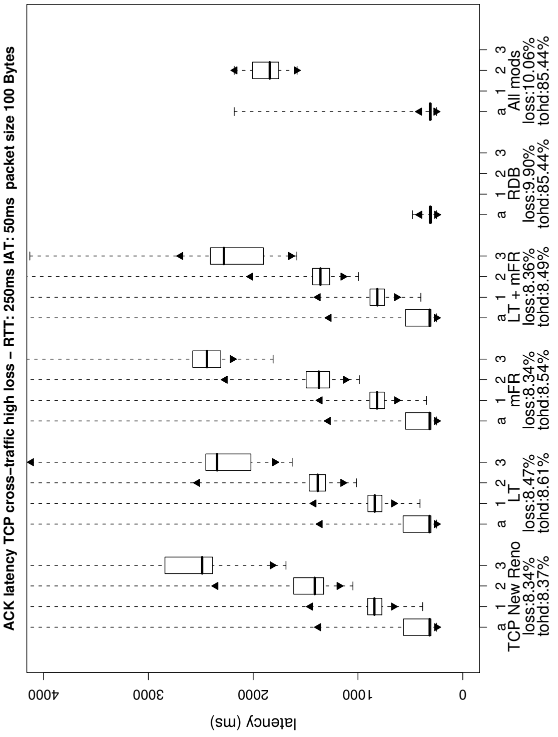

ACK latency TCP cross−traffic high loss − RTT: 250ms IAT: 150ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:9.43%
tohd:9.44%
a 1 2 3
LT
loss:9.27%
tohd:9.17%
a 1 2 3
mFR
loss:9.42%
tohd:9.41%
a 1 2 3
LT + mFR
loss:9.58%
tohd:9.45%
a 1 2 3
RDB
loss:9.86%
tohd:65.04%
a 1 2 3
All mods
loss:10.68%
tohd:65.20%

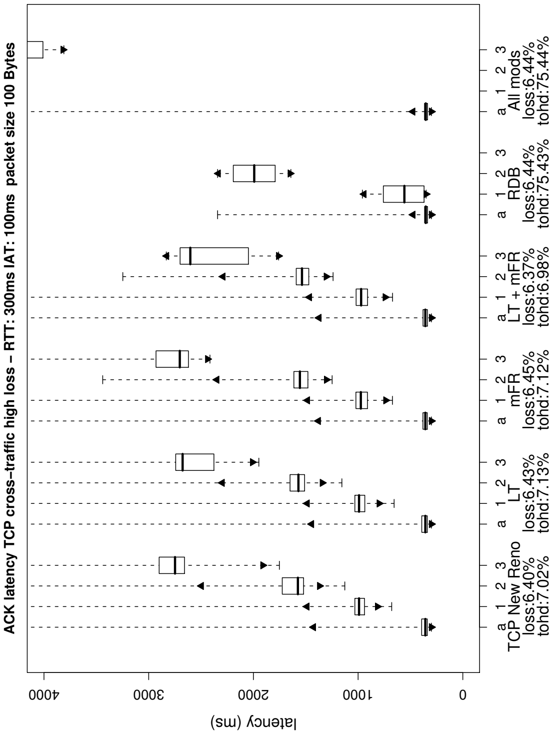

ACK latency TCP cross–traffic high loss – RTT: 250ms IAT: 250ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:9.66%
tohd:10.58%
a 1 2 3
LT
loss:9.80%
tohd:10.80%
a 1 2 3
mFR
loss:9.41%
tohd:10.38%
a 1 2 3
LT + mFR
loss:9.63%
tohd:10.56%
a 1 2 3
RDB
loss:10.63%
tohd:54.33%
a 1 2 3
All mods
loss:11.71%
tohd:54.74%

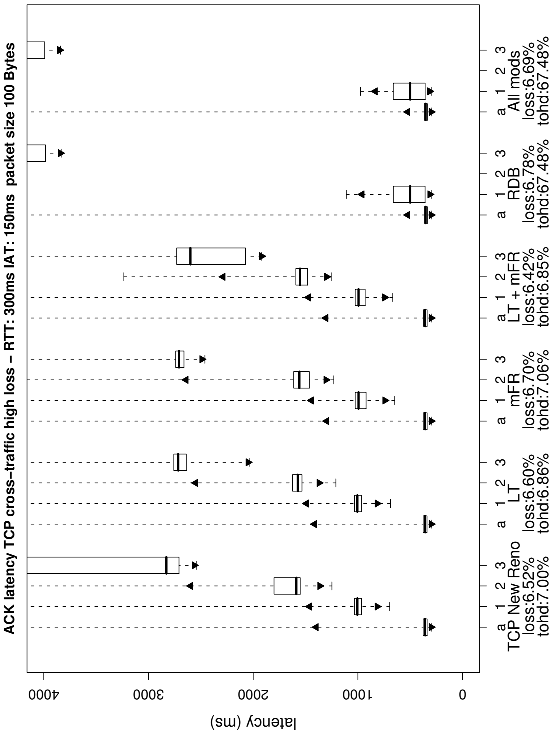

ACK latency TCP cross–traffic high loss – RTT: 250ms IAT: 500ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:11.68%
tohd:19.08%
a 1 2 3
LT
loss:10.47%
tohd:17.97%
a 1 2 3
mFR
loss:10.36%
tohd:17.40%
a 1 2 3
LT + mFR
loss:10.39%
tohd:18.02%
a 1 2 3
RDB
loss:11.41%
tohd:24.33%
a 1 2 3
All mods
loss:12.06%
tohd:25.45%

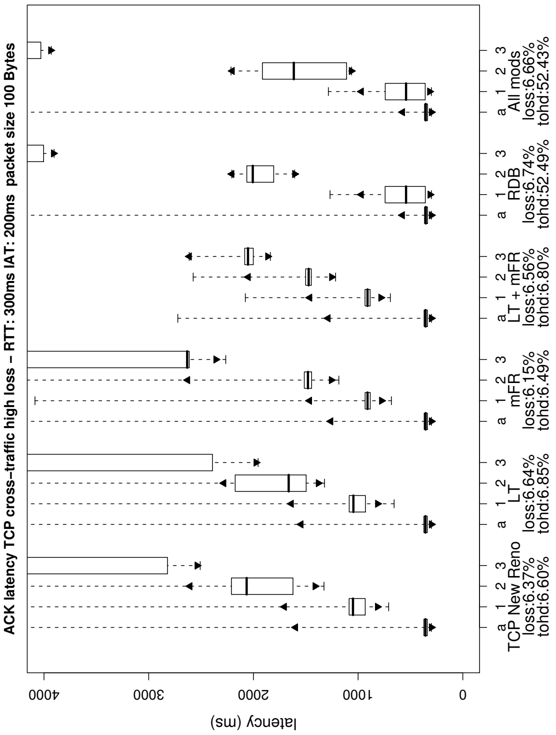

ACK latency TCP cross–traffic high loss – RTT: 250ms IAT: 200ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:9.99%
tohd:10.05%
a 1 2 3
LT
loss:9.77%
tohd:9.79%
a 1 2 3
mFR
loss:9.28%
tohd:9.20%
a 1 2 3
LT + mFR
loss:9.21%
tohd:9.21%
a 1 2 3
RDB
loss:10.72%
tohd:54.19%
a 1 2 3
All mods
loss:10.77%
tohd:54.24%

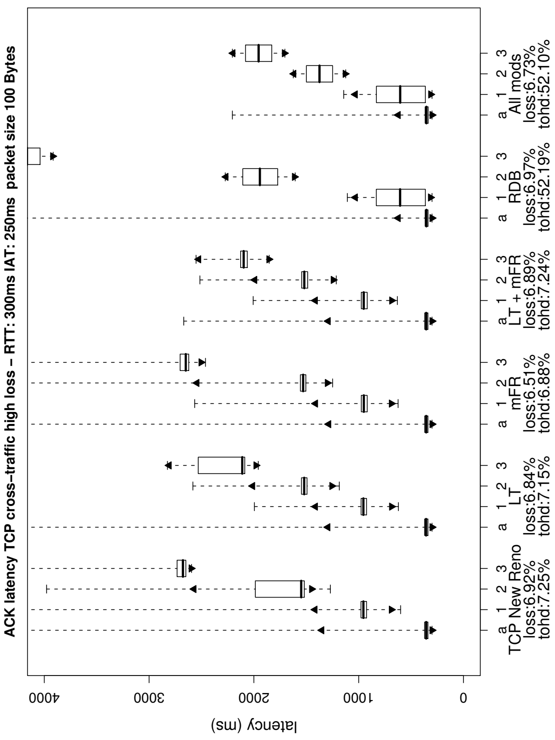

ACK latency TCP cross–traffic high loss – RTT: 250ms IAT: 25ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:7.97%
tohd:8.60%
a 1 2 3
LT
loss:7.95%
tohd:8.40%
a 1 2 3
mFR
loss:8.03%
tohd:8.38%
a 1 2 3
LT + mFR
loss:7.98%
tohd:8.39%
a 1 2 3
RDB
loss:9.42%
tohd:87.43%
a 1 2 3
All mods
loss:9.44%
tohd:87.39%

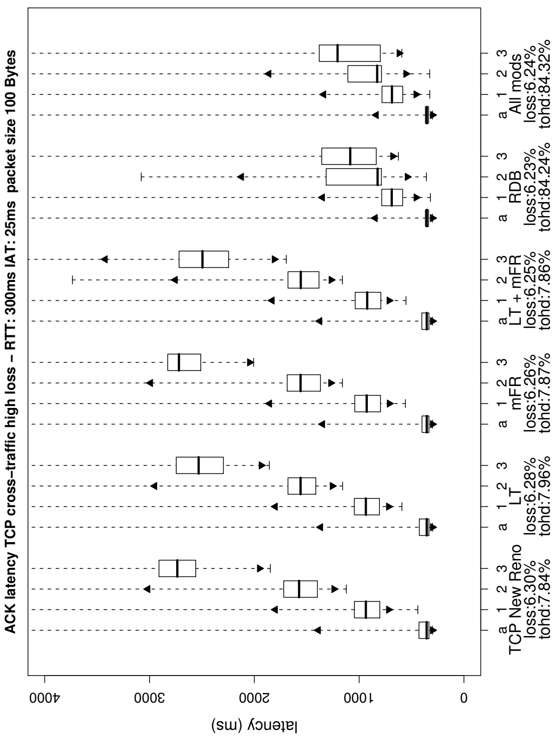

ACK latency TCP cross−traffic high loss − RTT: 250ms IAT: 50ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:8.34%
tohd:8.37%
a 1 2 3
LT
loss:8.47%
tohd:8.61%
a 1 2 3
mFR
loss:8.34%
tohd:8.54%
a 1 2 3
LT + mFR
loss:8.36%
tohd:8.49%
a 1 2 3
RDB
loss:9.90%
tohd:85.44%
a 1 2 3
All mods
loss:10.06%
tohd:85.44%

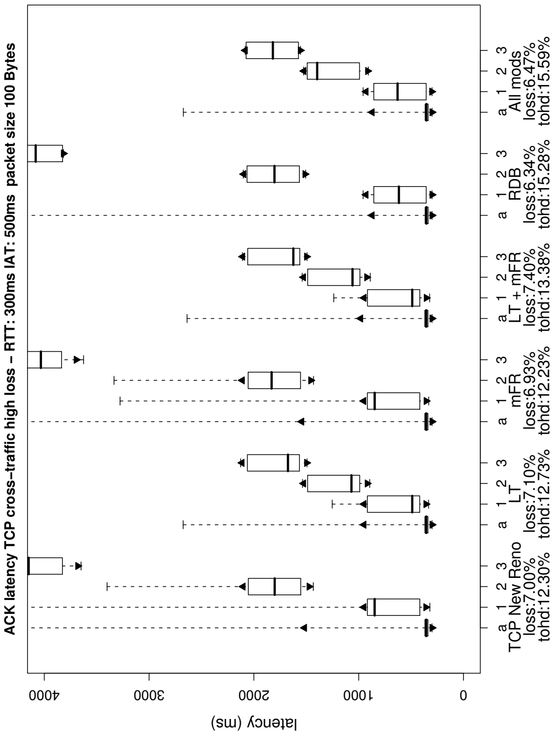

ACK latency TCP cross−traffic high loss − RTT: 300ms IAT: 100ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:6.40%
tohd:7.02%
a 1 2 3
LT
loss:6.43%
tohd:7.13%
a 1 2 3
mFR
loss:6.45%
tohd:7.12%
a 1 2 3
LT + mFR
loss:6.37%
tohd:6.98%
a 1 2 3
RDB
loss:6.44%
tohd:75.43%
a 1 2 3
All mods
loss:6.44%
tohd:75.44%

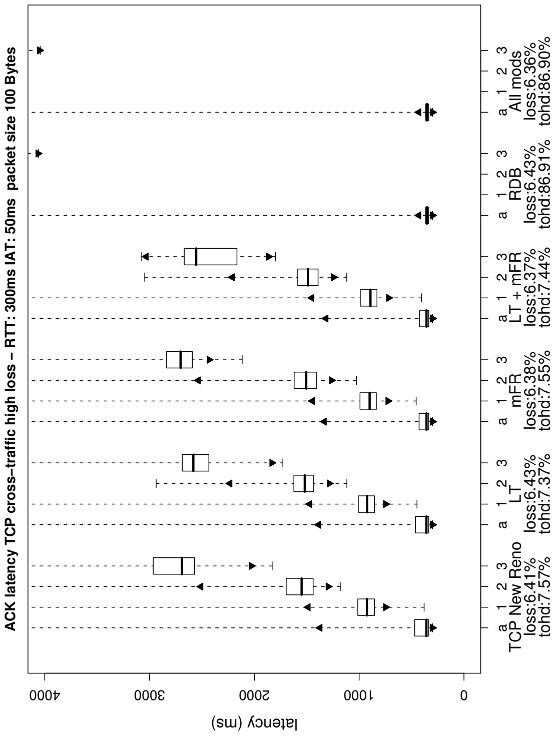

ACK latency TCP cross−traffic high loss − RTT: 300ms IAT: 150ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:6.52%
tohd:7.00%
a 1 2 3
LT
loss:6.60%
tohd:6.86%
a 1 2 3
mFR
loss:6.70%
tohd:7.06%
a 1 2 3
LT + mFR
loss:6.42%
tohd:6.85%
a 1 2 3
RDB
loss:6.78%
tohd:67.48%
a 1 2 3
All mods
loss:6.69%
tohd:67.48%

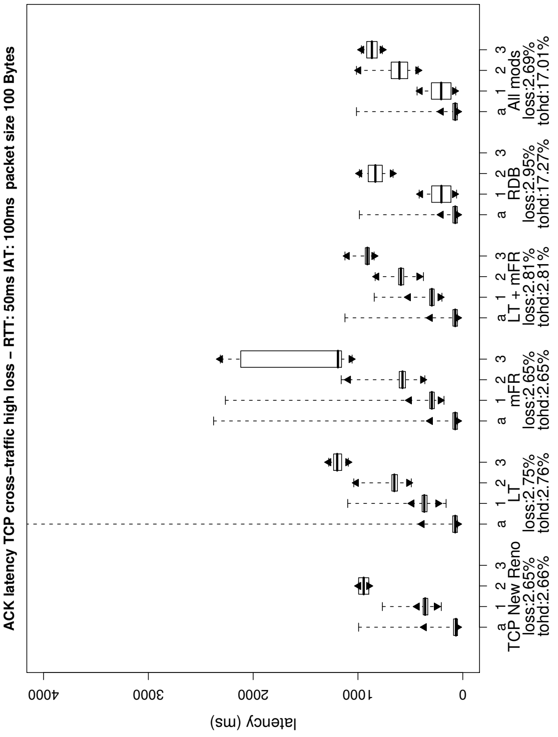

ACK latency TCP cross−traffic high loss − RTT: 300ms IAT: 200ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:6.37%
tohd:6.60%
a 1 2 3
LT
loss:6.64%
tohd:6.85%
a 1 2 3
mFR
loss:6.15%
tohd:6.49%
a 1 2 3
LT + mFR
loss:6.56%
tohd:6.80%
a 1 2 3
RDB
loss:6.74%
tohd:52.49%
a 1 2 3
All mods
loss:6.66%
tohd:52.43%

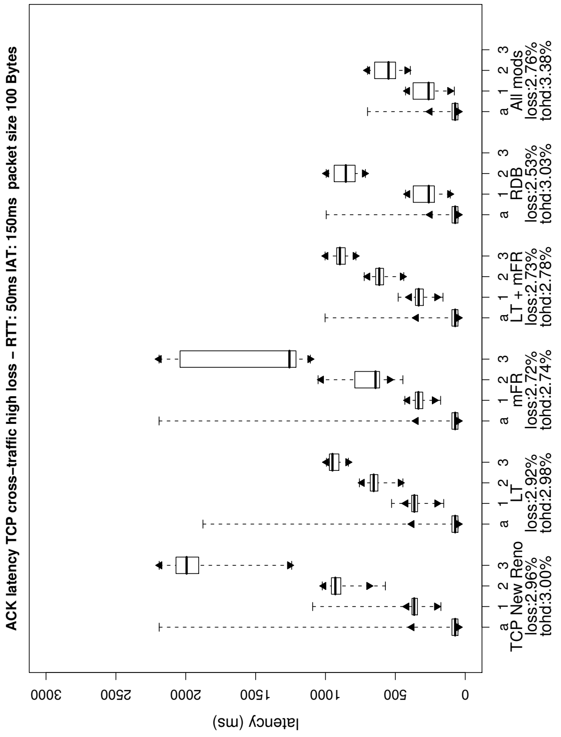

ACK latency TCP cross–traffic high loss – RTT: 300ms IAT: 250ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:6.92%
tohd:7.25%
a 1 2 3
LT
loss:6.84%
tohd:7.15%
a 1 2 3
mFR
loss:6.51%
tohd:6.88%
a 1 2 3
LT + mFR
loss:6.89%
tohd:7.24%
a 1 2 3
RDB
loss:6.97%
tohd:52.19%
a 1 2 3
All mods
loss:6.73%
tohd:52.10%

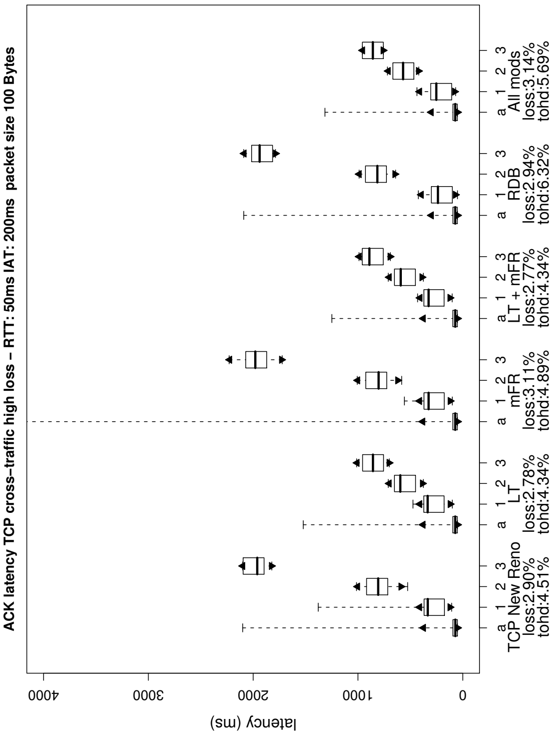

ACK latency TCP cross–traffic high loss – RTT: 300ms IAT: 25ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:6.30%
tohd:7.84%
a 1 2 3
LT
loss:6.28%
tohd:7.96%
a 1 2 3
mFR
loss:6.26%
tohd:7.87%
a 1 2 3
LT + mFR
loss:6.25%
tohd:7.86%
a 1 2 3
RDB
loss:6.23%
tohd:84.24%
a 1 2 3
All mods
loss:6.24%
tohd:84.32%

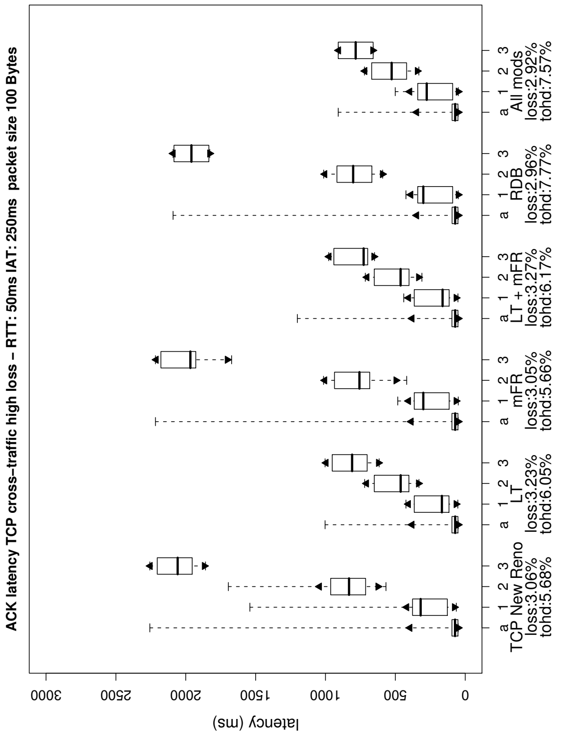

ACK latency TCP cross–traffic high loss – RTT: 300ms IAT: 500ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:7.00%
tohd:12.30%
a 1 2 3
LT
loss:7.10%
tohd:12.73%
a 1 2 3
mFR
loss:6.93%
tohd:12.23%
a 1 2 3
LT + mFR
loss:7.40%
tohd:13.38%
a 1 2 3
RDB
loss:6.34%
tohd:15.28%
a 1 2 3
All mods
loss:6.47%
tohd:15.59%

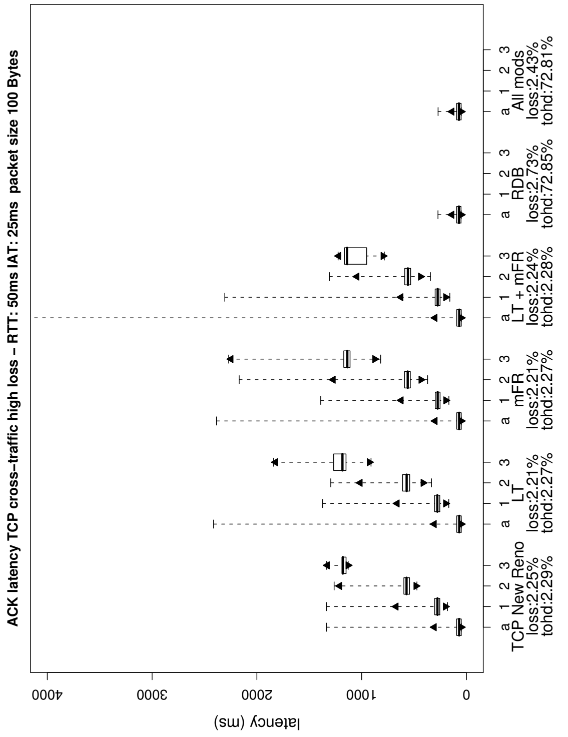

ACK latency TCP cross–traffic high loss – RTT: 300ms IAT: 50ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:6.41%
tohd:7.57%
a 1 2 3
LT
loss:6.43%
tohd:7.37%
a 1 2 3
mFR
loss:6.38%
tohd:7.55%
a 1 2 3
LT + mFR
loss:6.37%
tohd:7.44%
a 1 2 3
RDB
loss:6.43%
tohd:86.91%
a 1 2 3
All mods
loss:6.36%
tohd:86.90%

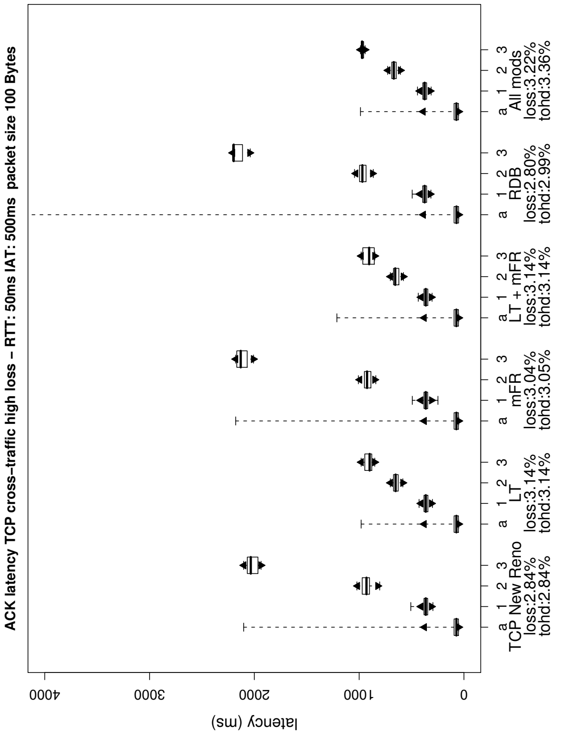

ACK latency TCP cross−traffic high loss − RTT: 50ms IAT: 100ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:2.65%
tohd:2.66%
a 1 2 3
LT
loss:2.75%
tohd:2.76%
a 1 2 3
mFR
loss:2.65%
tohd:2.65%
a 1 2 3
LT + mFR
loss:2.81%
tohd:2.81%
a 1 2 3
RDB
loss:2.95%
tohd:17.27%
a 1 2 3
All mods
loss:2.69%
tohd:17.01%

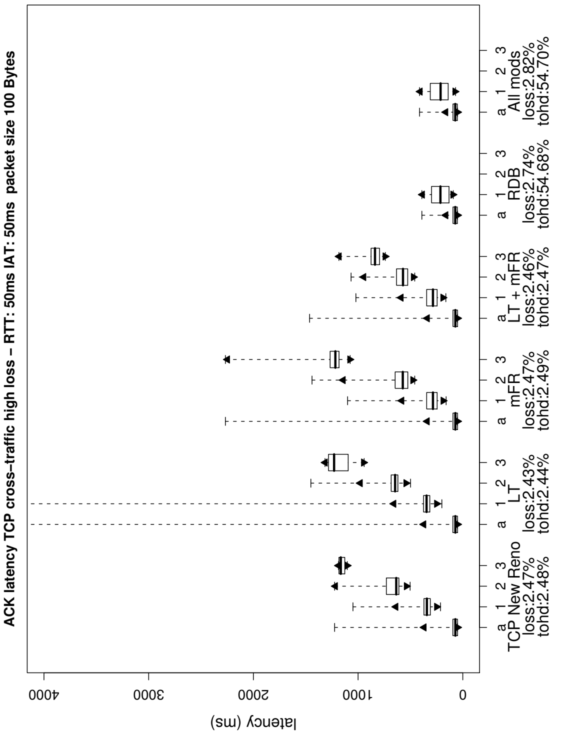

ACK latency TCP cross−traffic high loss − RTT: 50ms IAT: 150ms packet size 100 Bytes
latency (ms)
0
500
1000
1500
2000
2500
3000
a 1 2 3
TCP New Reno
loss:2.96%
tohd:3.00%
a 1 2 3
LT
loss:2.92%
tohd:2.98%
a 1 2 3
mFR
loss:2.72%
tohd:2.74%
a 1 2 3
LT + mFR
loss:2.73%
tohd:2.78%
a 1 2 3
RDB
loss:2.53%
tohd:3.03%
a 1 2 3
All mods
loss:2.76%
tohd:3.38%


ACK latency TCP cross−traffic high loss − RTT: 50ms IAT: 200ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:2.90%
tohd:4.51%
a 1 2 3
LT
loss:2.78%
tohd:4.34%
a 1 2 3
mFR
loss:3.11%
tohd:4.89%
a 1 2 3
LT + mFR
loss:2.77%
tohd:4.34%
a 1 2 3
RDB
loss:2.94%
tohd:6.32%
a 1 2 3
All mods
loss:3.14%
tohd:5.69%

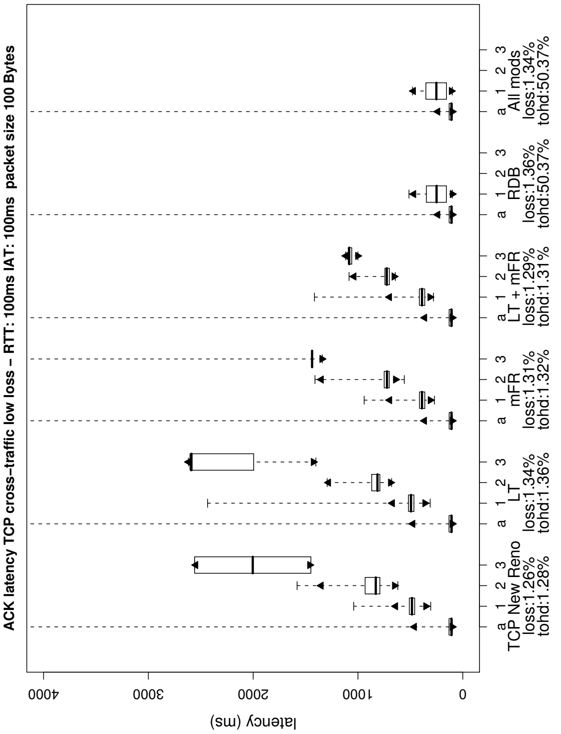

ACK latency TCP cross−traffic high loss − RTT: 50ms IAT: 250ms packet size 100 Bytes
latency (ms)
0
500
1000
1500
2000
2500
3000
a 1 2 3
TCP New Reno
loss:3.06%
tohd:5.68%
a 1 2 3
LT
loss:3.23%
tohd:6.05%
a 1 2 3
mFR
loss:3.05%
tohd:5.66%
a 1 2 3
LT + mFR
loss:3.27%
tohd:6.17%
a 1 2 3
RDB
loss:2.96%
tohd:7.77%
a 1 2 3
All mods
loss:2.92%
tohd:7.57%

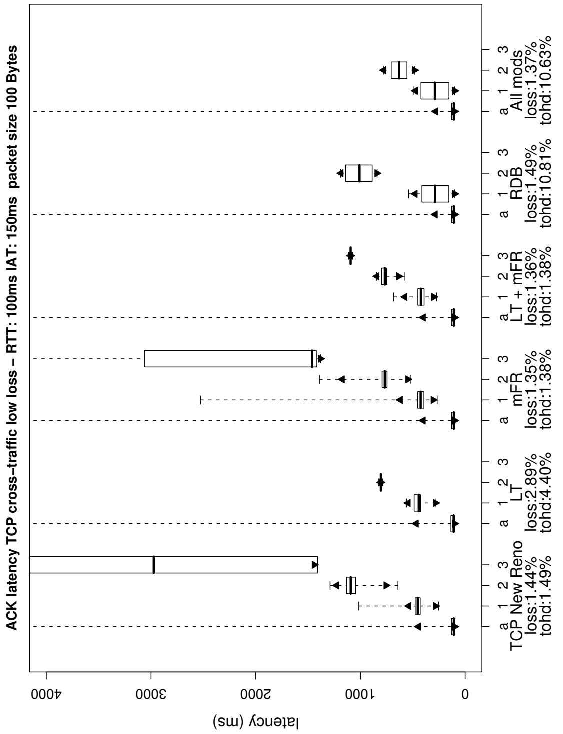

ACK latency TCP cross–traffic high loss – RTT: 50ms IAT: 25ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:2.25%
tohd:2.29%
a 1 2 3
LT
loss:2.21%
tohd:2.27%
a 1 2 3
mFR
loss:2.21%
tohd:2.27%
a 1 2 3
LT + mFR
loss:2.24%
tohd:2.28%
a 1 2 3
RDB
loss:2.73%
tohd:72.85%
a 1 2 3
All mods
loss:2.43%
tohd:72.81%

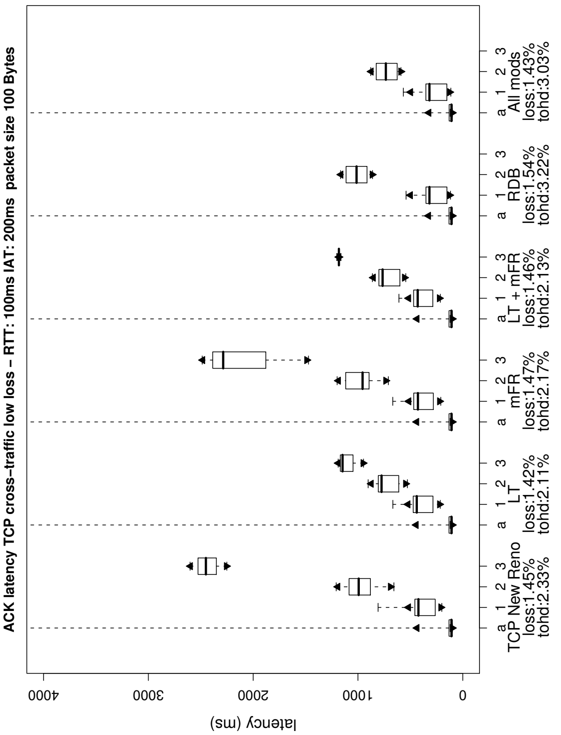

ACK latency TCP cross–traffic high loss – RTT: 50ms IAT: 500ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:2.84%
tohd:2.84%
a 1 2 3
LT
loss:3.14%
tohd:3.14%
a 1 2 3
mFR
loss:3.04%
tohd:3.05%
a 1 2 3
LT + mFR
loss:3.14%
tohd:3.14%
a 1 2 3
RDB
loss:2.80%
tohd:2.99%
a 1 2 3
All mods
loss:3.22%
tohd:3.36%

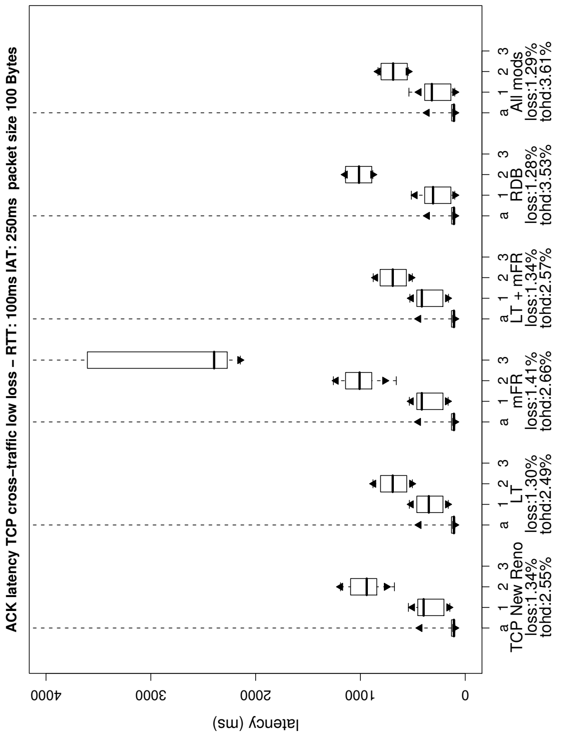

ACK latency TCP cross–traffic high loss – RTT: 50ms IAT: 50ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:2.47%
tohd:2.48%
a 1 2 3
LT
loss:2.43%
tohd:2.44%
a 1 2 3
mFR
loss:2.47%
tohd:2.49%
a 1 2 3
LT + mFR
loss:2.46%
tohd:2.47%
a 1 2 3
RDB
loss:2.74%
tohd:54.68%
a 1 2 3
All mods
loss:2.82%
tohd:54.70%

C.3 Cross-traffic loss - low loss rate

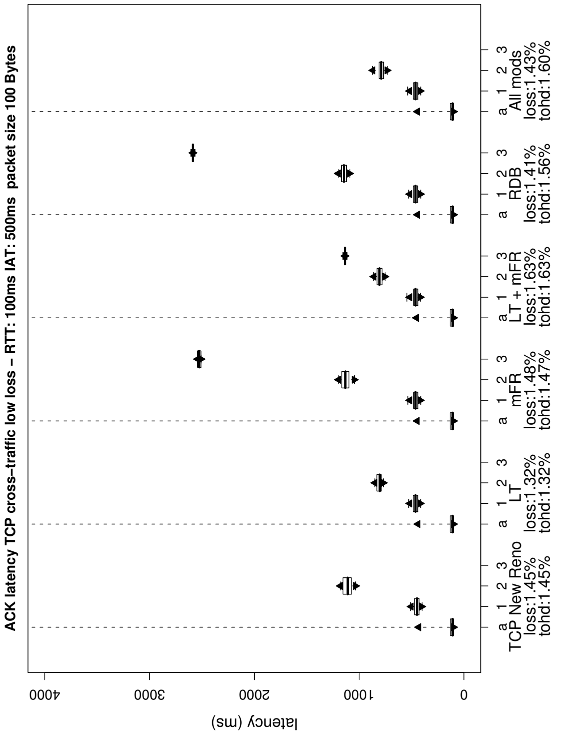


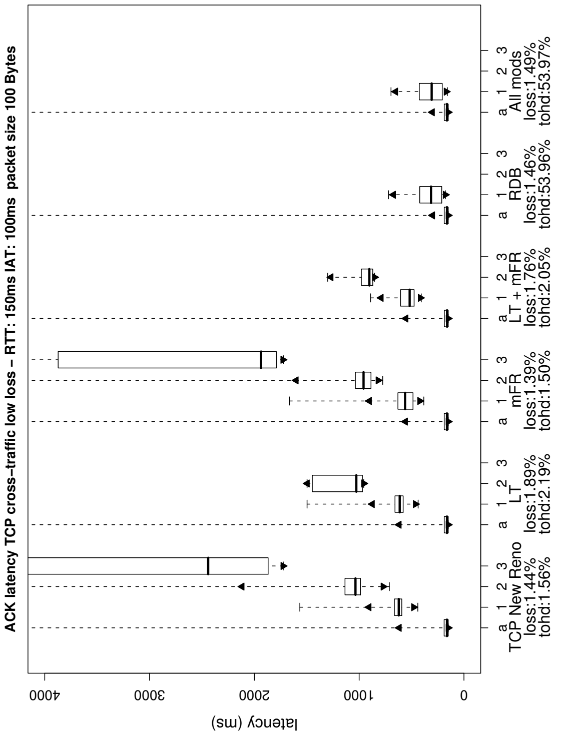


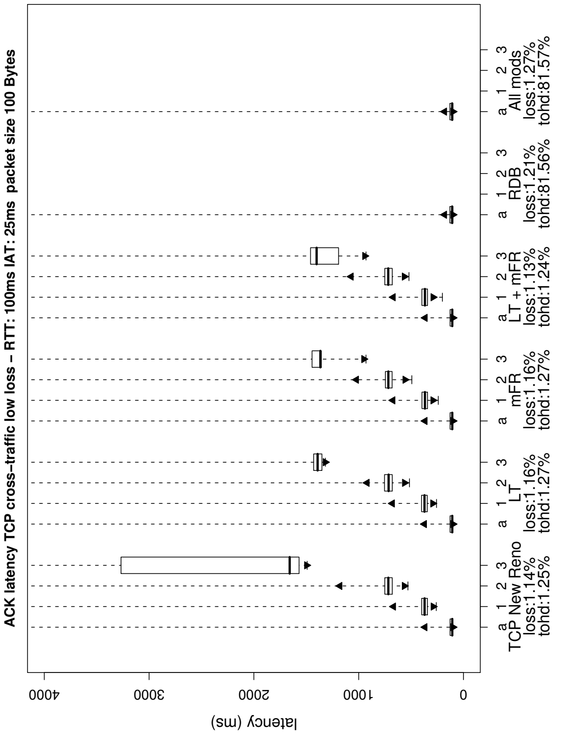


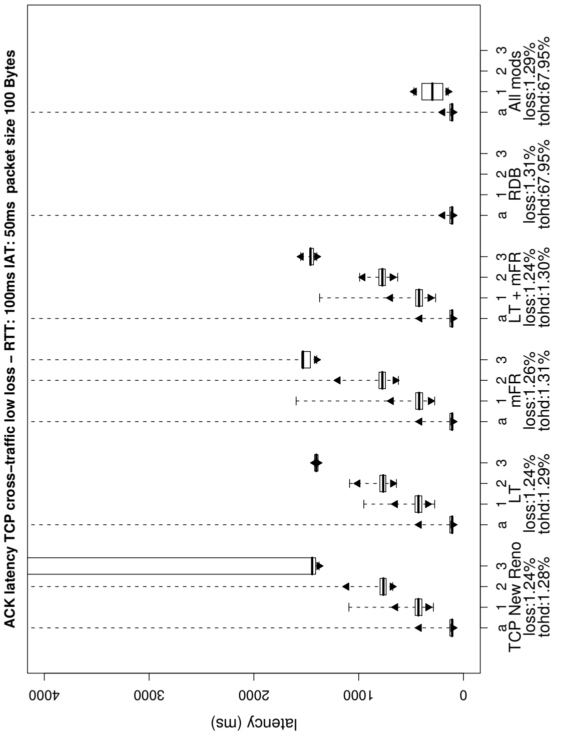


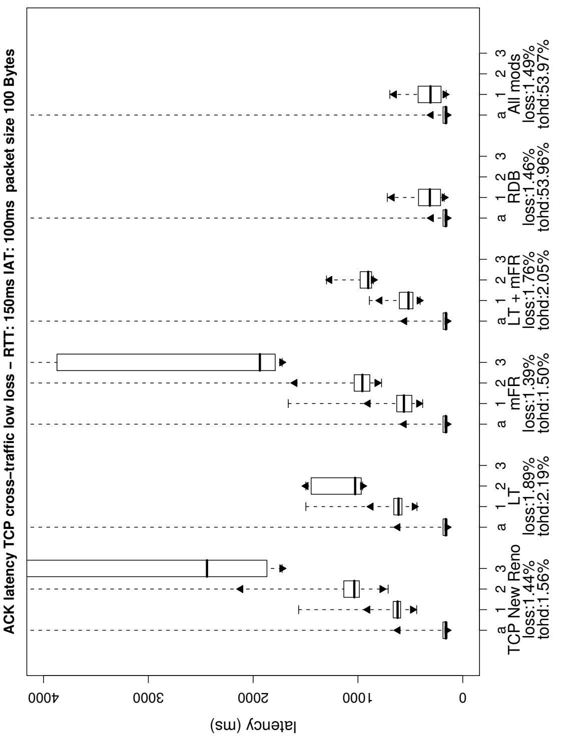
]

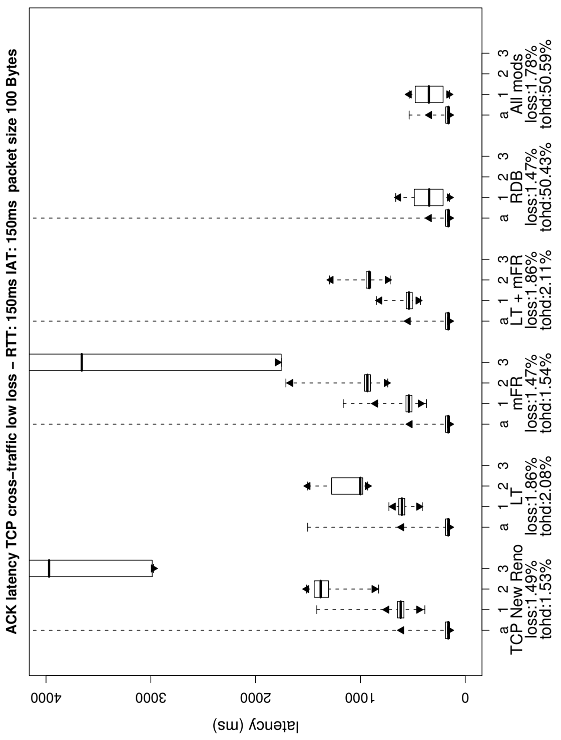

ACK latency TCP cross–traffic low loss – RTT: 100ms IAT: 500ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:1.45%
tohd:1.45%
a 1 2 3
LT
loss:1.32%
tohd:1.32%
a 1 2 3
mFR
loss:1.48%
tohd:1.47%
a 1 2 3
LT + mFR
loss:1.63%
tohd:1.63%
a 1 2 3
RDB
loss:1.41%
tohd:1.56%
a 1 2 3
All mods
loss:1.43%
tohd:1.60%

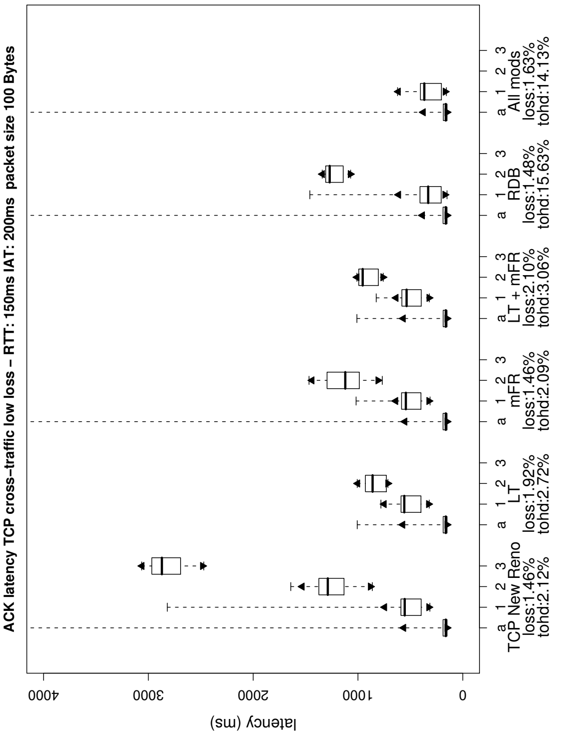

ACK latency TCP cross–traffic low loss – RTT: 150ms IAT: 100ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:1.44%
tohd:1.56%
a 1 2 3
LT
loss:1.89%
tohd:2.19%
a 1 2 3
mFR
loss:1.39%
tohd:1.50%
a 1 2 3
LT + mFR
loss:1.76%
tohd:2.05%
a 1 2 3
RDB
loss:1.46%
tohd:53.96%
a 1 2 3
All mods
loss:1.49%
tohd:53.97%

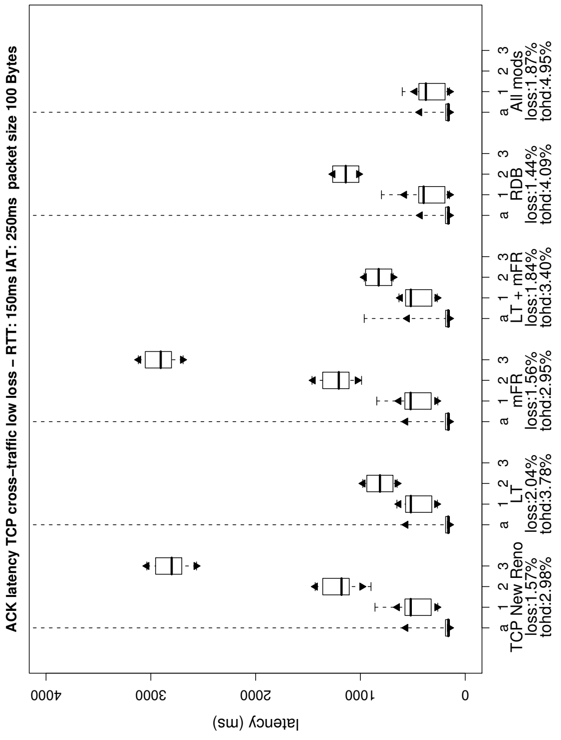

ACK latency TCP cross–traffic low loss – RTT: 100ms IAT: 25ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:1.14%
tohd:1.25%
a 1 2 3
LT
loss:1.16%
tohd:1.27%
a 1 2 3
mFR
loss:1.16%
tohd:1.27%
a 1 2 3
LT + mFR
loss:1.13%
tohd:1.24%
a 1 2 3
RDB
loss:1.21%
tohd:81.56%
a 1 2 3
All mods
loss:1.27%
tohd:81.57%

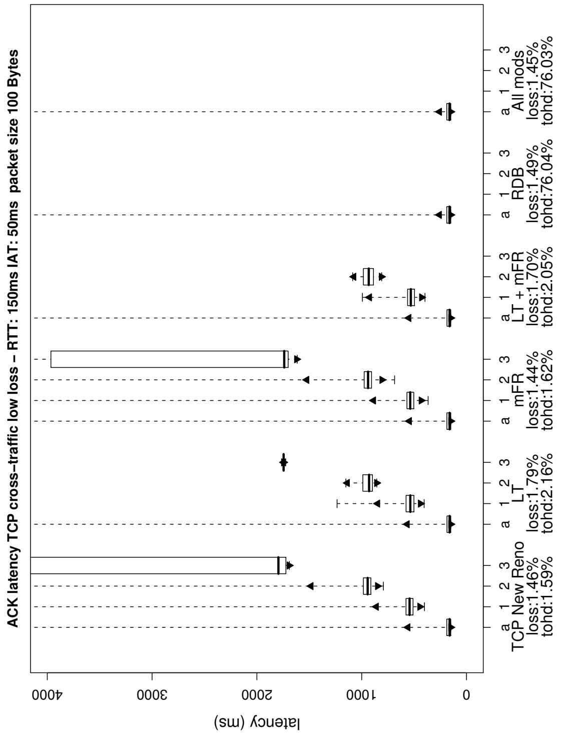

ACK latency TCP cross–traffic low loss – RTT: 100ms IAT: 50ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:1.24%
tohd:1.28%
a 1 2 3
LT
loss:1.24%
tohd:1.29%
a 1 2 3
mFR
loss:1.26%
tohd:1.31%
a 1 2 3
LT + mFR
loss:1.24%
tohd:1.30%
a 1 2 3
RDB
loss:1.31%
tohd:67.95%
a 1 2 3
All mods
loss:1.29%
tohd:67.95%

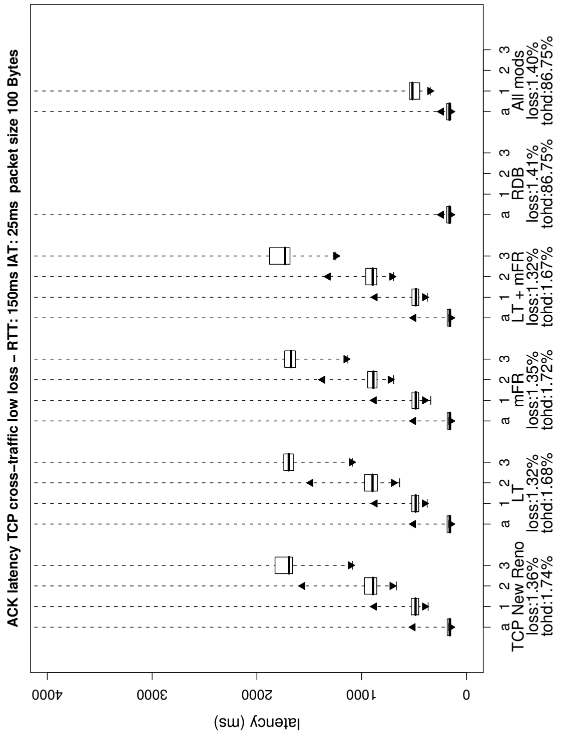

ACK latency TCP cross−traffic low loss − RTT: 150ms IAT: 100ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:1.44%
tohd:1.56%
a 1 2 3
LT
loss:1.89%
tohd:2.19%
a 1 2 3
mFR
loss:1.39%
tohd:1.50%
a 1 2 3
LT + mFR
loss:1.76%
tohd:2.05%
a 1 2 3
RDB
loss:1.46%
tohd:53.96%
a 1 2 3
All mods
loss:1.49%
tohd:53.97%

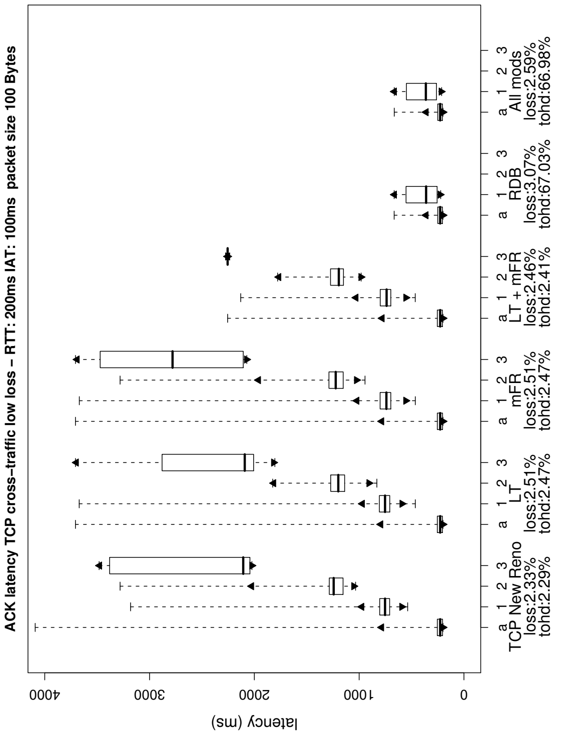

ACK latency TCP cross−traffic low loss − RTT: 150ms IAT: 150ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:1.49%
tohd:1.53%
a 1 2 3
LT
loss:1.86%
tohd:2.08%
a 1 2 3
mFR
loss:1.47%
tohd:1.54%
a 1 2 3
LT + mFR
loss:1.86%
tohd:2.11%
a 1 2 3
RDB
loss:1.47%
tohd:50.43%
a 1 2 3
All mods
loss:1.78%
tohd:50.59%

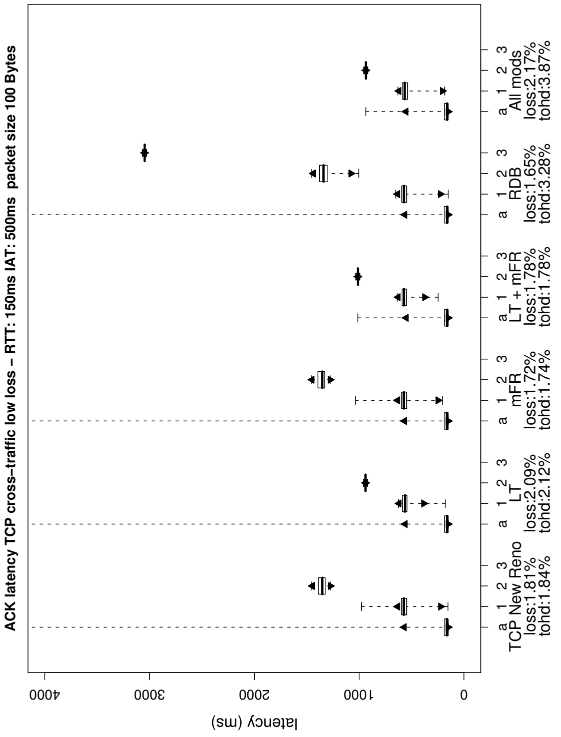

ACK latency TCP cross−traffic low loss − RTT: 150ms IAT: 200ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:1.46%
tohd:2.12%
a 1 2 3
LT
loss:1.92%
tohd:2.72%
a 1 2 3
mFR
loss:1.46%
tohd:2.09%
a 1 2 3
LT + mFR
loss:2.10%
tohd:3.06%
a 1 2 3
RDB
loss:1.48%
tohd:15.63%
a 1 2 3
All mods
loss:1.63%
tohd:14.13%

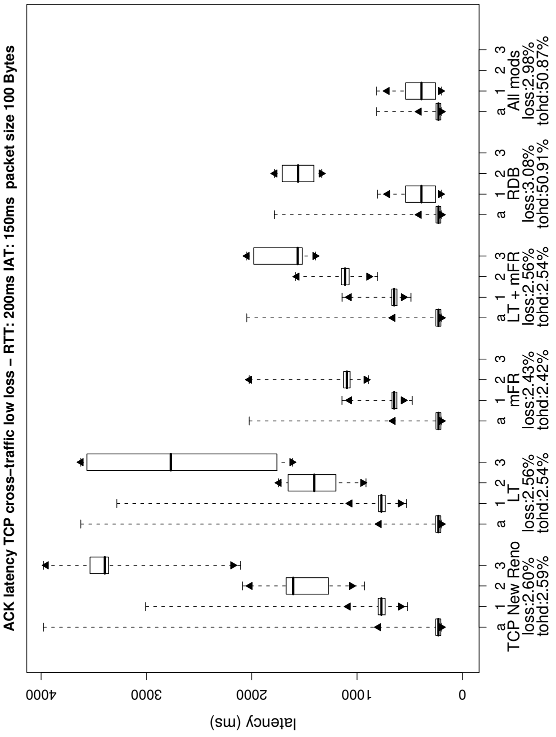

ACK latency TCP cross−traffic low loss − RTT: 150ms IAT: 250ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:1.57%
tohd:2.98%
a 1 2 3
LT
loss:2.04%
tohd:3.78%
a 1 2 3
mFR
loss:1.56%
tohd:2.95%
a 1 2 3
LT + mFR
loss:1.84%
tohd:3.40%
a 1 2 3
RDB
loss:1.44%
tohd:4.09%
a 1 2 3
All mods
loss:1.87%
tohd:4.95%

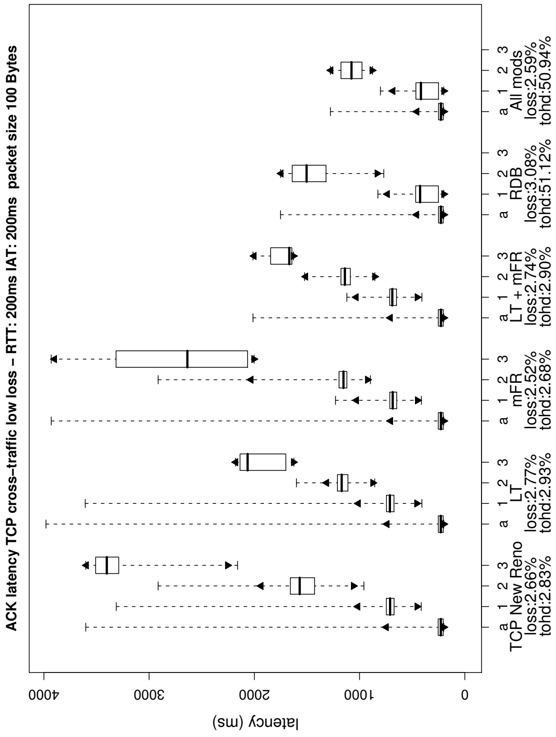

ACK latency TCP cross−traffic low loss − RTT: 150ms IAT: 50ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:1.46%
tohd:1.59%
a 1 2 3
LT
loss:1.79%
tohd:2.16%
a 1 2 3
mFR
loss:1.44%
tohd:1.62%
a 1 2 3
LT + mFR
loss:1.70%
tohd:2.05%
a 1 2 3
RDB
loss:1.49%
tohd:76.04%
a 1 2 3
All mods
loss:1.45%
tohd:76.03%

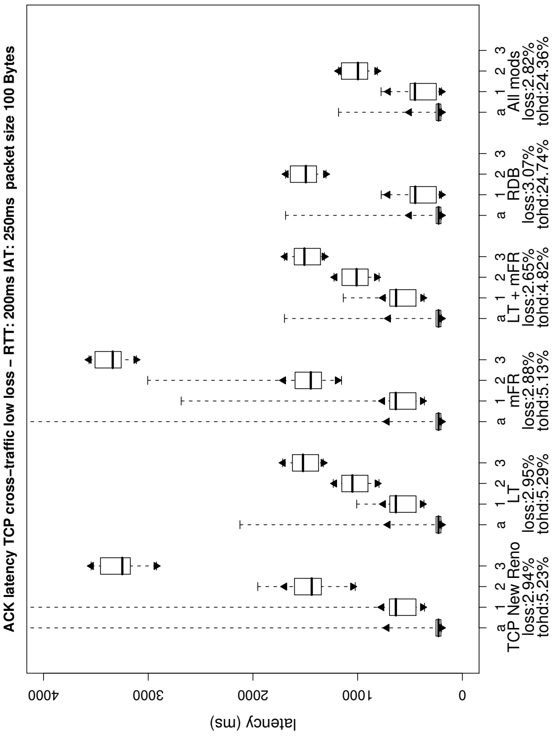

ACK latency TCP cross−traffic low loss − RTT: 150ms IAT: 25ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:1.36%
tohd:1.74%
a 1 2 3
LT
loss:1.32%
tohd:1.68%
a 1 2 3
mFR
loss:1.35%
tohd:1.72%
a 1 2 3
LT + mFR
loss:1.32%
tohd:1.67%
a 1 2 3
RDB
loss:1.41%
tohd:86.75%
a 1 2 3
All mods
loss:1.40%
tohd:86.75%

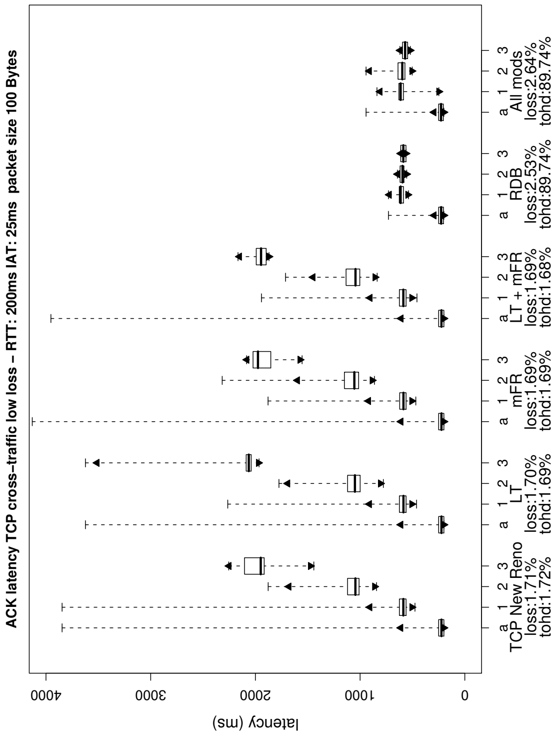

ACK latency TCP cross−traffic low loss − RTT: 200ms IAT: 100ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:2.33%
tohd:2.29%
a 1 2 3
LT
loss:2.51%
tohd:2.47%
a 1 2 3
mFR
loss:2.51%
tohd:2.47%
a 1 2 3
LT + mFR
loss:2.46%
tohd:2.41%
a 1 2 3
RDB
loss:3.07%
tohd:67.03%
a 1 2 3
All mods
loss:2.59%
tohd:66.98%

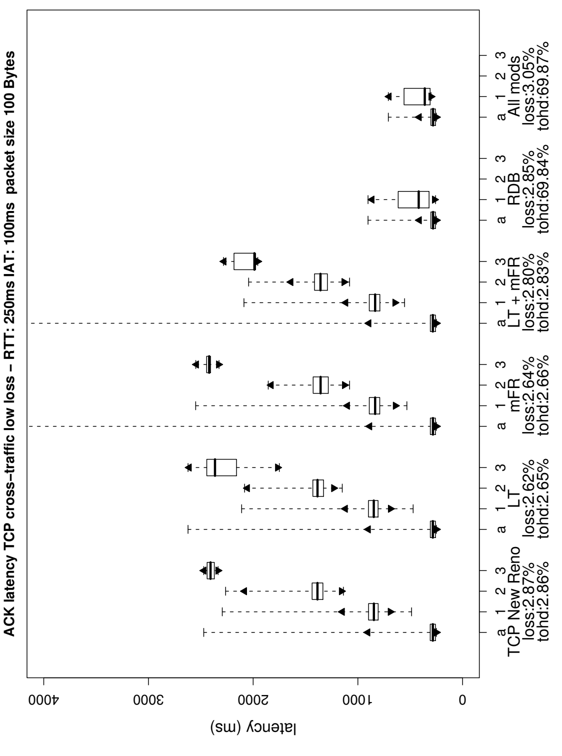

ACK latency TCP cross−traffic low loss − RTT: 150ms IAT: 500ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:1.81%
tohd:1.84%
a 1 2 3
LT
loss:2.09%
tohd:2.12%
a 1 2 3
mFR
loss:1.72%
tohd:1.74%
a 1 2 3
LT + mFR
loss:1.78%
tohd:1.78%
a 1 2 3
RDB
loss:1.65%
tohd:3.28%
a 1 2 3
All mods
loss:2.17%
tohd:3.87%

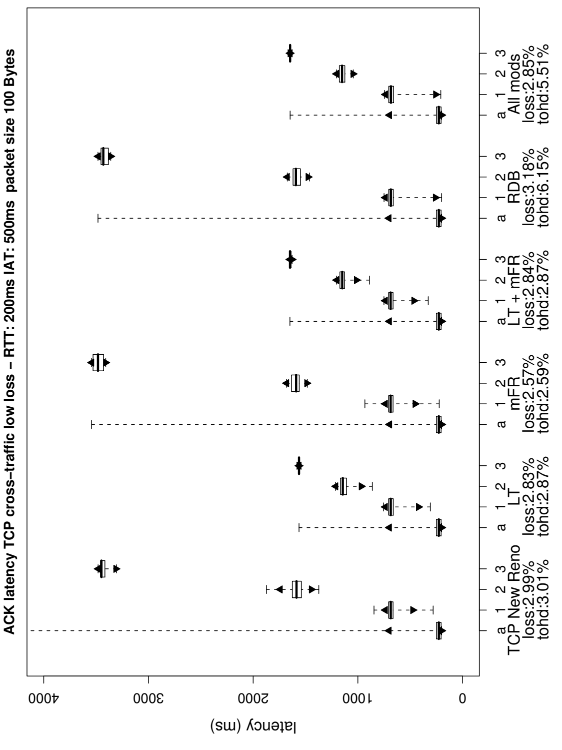

ACK latency TCP cross−traffic low loss − RTT: 200ms IAT: 150ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:2.60%
tohd:2.59%
a 1 2 3
LT
loss:2.56%
tohd:2.54%
a 1 2 3
mFR
loss:2.43%
tohd:2.42%
a 1 2 3
LT + mFR
loss:2.56%
tohd:2.54%
a 1 2 3
RDB
loss:3.08%
tohd:50.91%
a 1 2 3
All mods
loss:2.98%
tohd:50.87%

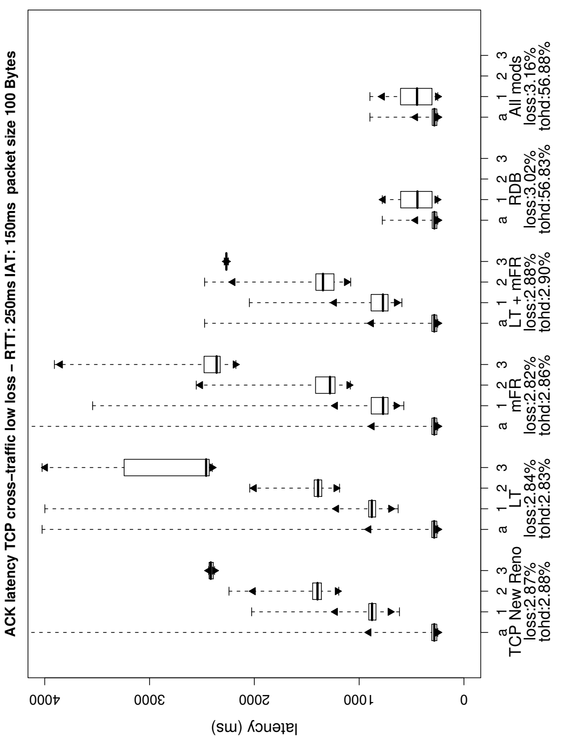

ACK latency TCP cross−traffic low loss − RTT: 200ms IAT: 200ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:2.66%
tohd:2.83%
a 1 2 3
LT
loss:2.77%
tohd:2.93%
a 1 2 3
mFR
loss:2.52%
tohd:2.68%
a 1 2 3
LT + mFR
loss:2.74%
tohd:2.90%
a 1 2 3
RDB
loss:3.08%
tohd:51.12%
a 1 2 3
All mods
loss:2.59%
tohd:50.94%

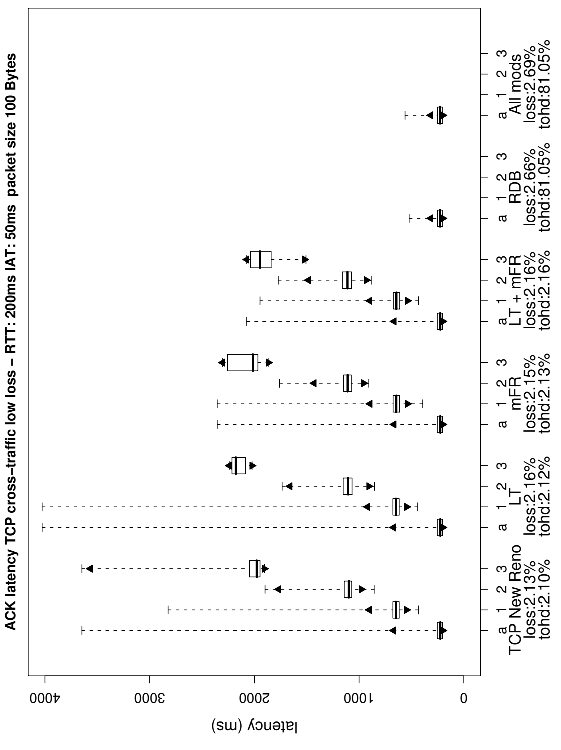

ACK latency TCP cross−traffic low loss − RTT: 200ms IAT: 250ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:2.94%
tohd:5.23%
a 1 2 3
LT
loss:2.95%
tohd:5.29%
a 1 2 3
mFR
loss:2.88%
tohd:5.13%
a 1 2 3
LT + mFR
loss:2.65%
tohd:4.82%
a 1 2 3
RDB
loss:3.07%
tohd:24.74%
a 1 2 3
All mods
loss:2.82%
tohd:24.36%

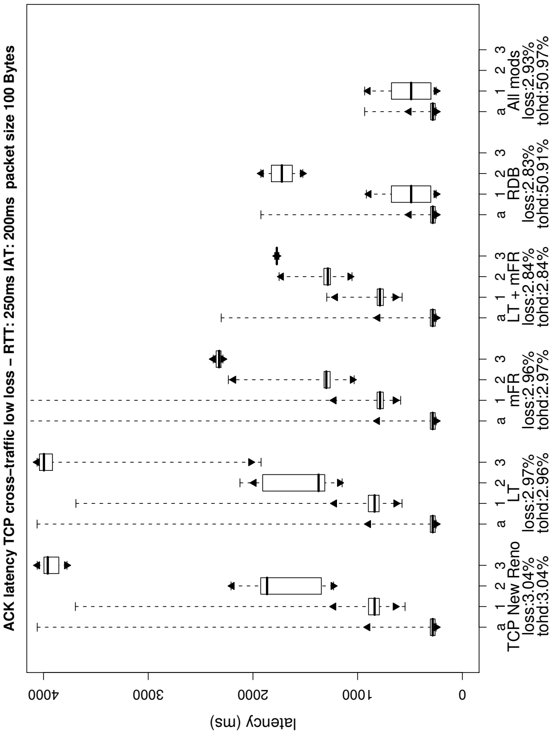

ACK latency TCP cross−traffic low loss − RTT: 200ms IAT: 25ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:1.71%
tohd:1.72%
a 1 2 3
LT
loss:1.70%
tohd:1.69%
a 1 2 3
mFR
loss:1.69%
tohd:1.69%
a 1 2 3
LT + mFR
loss:1.69%
tohd:1.68%
a 1 2 3
RDB
loss:2.53%
tohd:89.74%
a 1 2 3
All mods
loss:2.64%
tohd:89.74%

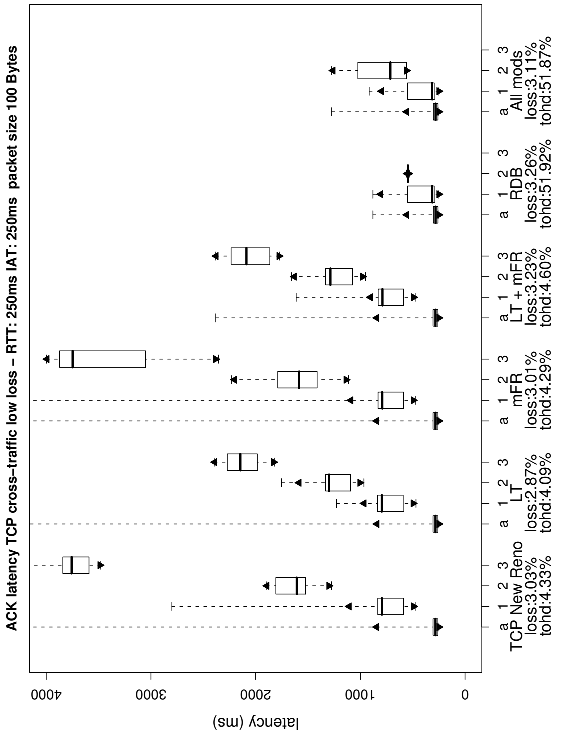

ACK latency TCP cross−traffic low loss − RTT: 250ms IAT: 100ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:2.87%
tohd:2.86%
LT
loss:2.62%
tohd:2.65%
mFR
loss:2.64%
tohd:2.66%
LT + mFR
loss:2.80%
tohd:2.83%
RDB
loss:2.85%
tohd:69.84%
All mods
loss:3.05%
tohd:69.87%

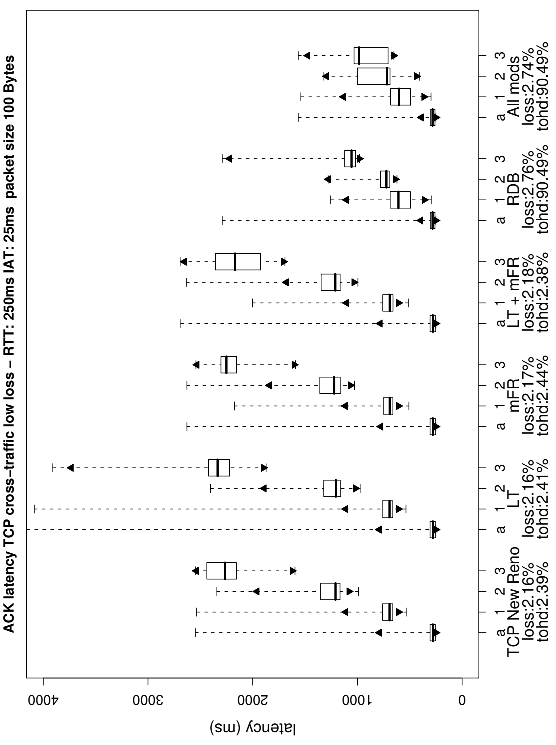

ACK latency TCP cross−traffic low loss − RTT: 200ms IAT: 500ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:2.99%
tohd:3.01%
LT
loss:2.83%
tohd:2.87%
mFR
loss:2.57%
tohd:2.59%
LT + mFR
loss:2.84%
tohd:2.87%
RDB
loss:3.18%
tohd:6.15%
All mods
loss:2.85%
tohd:5.51%

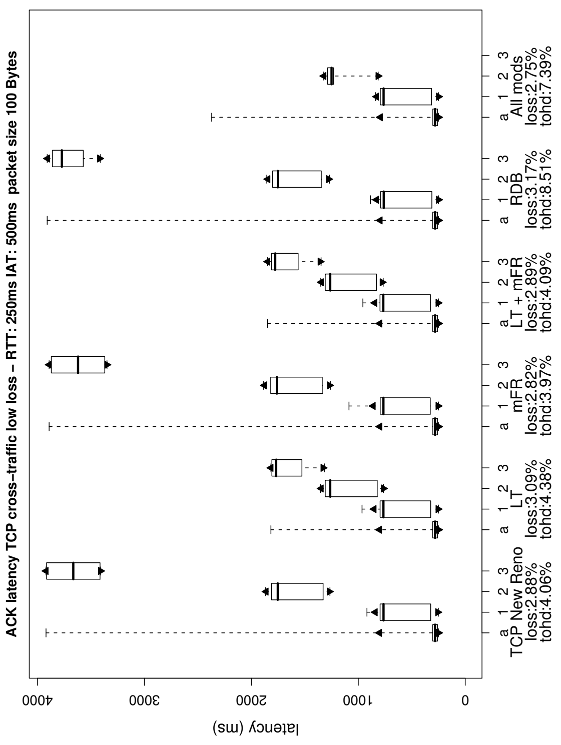

ACK latency TCP cross−traffic low loss − RTT: 250ms IAT: 150ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:2.87%
tohd:2.88%
LT
loss:2.84%
tohd:2.83%
mFR
loss:2.82%
tohd:2.86%
LT + mFR
loss:2.88%
tohd:2.90%
RDB
loss:3.02%
tohd:56.83%
All mods
loss:3.16%
tohd:56.88%

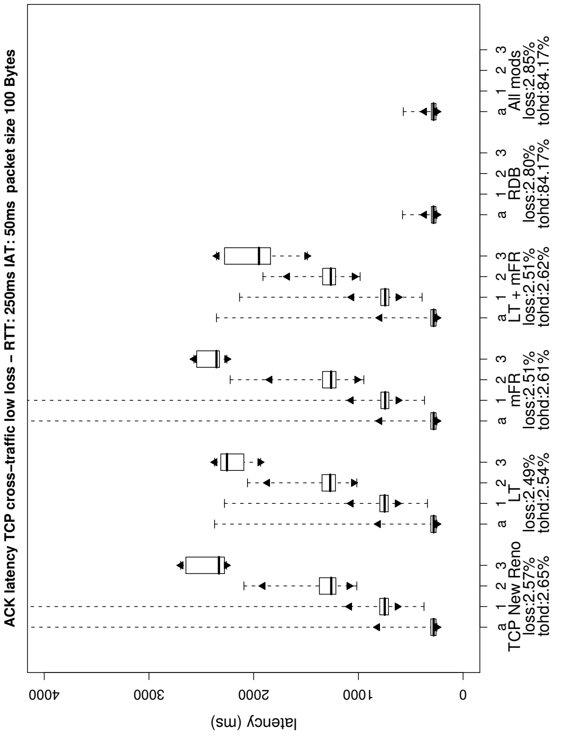

ACK latency TCP cross−traffic low loss − RTT: 200ms IAT: 50ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:2.13%
tohd:2.10%
LT
loss:2.16%
tohd:2.12%
mFR
loss:2.15%
tohd:2.13%
LT + mFR
loss:2.16%
tohd:2.16%
RDB
loss:2.66%
tohd:81.05%
All mods
loss:2.69%
tohd:81.05%

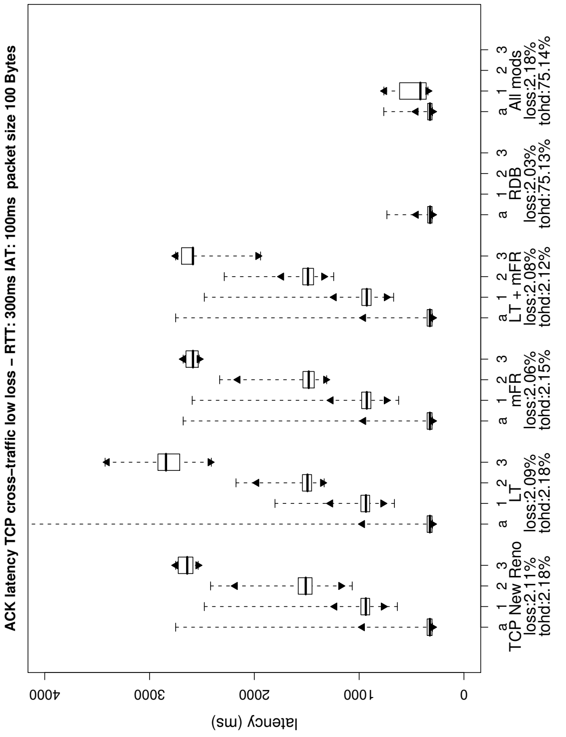

ACK latency TCP cross−traffic low loss − RTT: 250ms IAT: 200ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:3.04%
tohd:3.04%
a 1 2 3
LT
loss:2.97%
tohd:2.96%
a 1 2 3
mFR
loss:2.96%
tohd:2.97%
a 1 2 3
LT + mFR
loss:2.84%
tohd:2.84%
a 1 2 3
RDB
loss:2.83%
tohd:50.91%
a 1 2 3
All mods
loss:2.93%
tohd:50.97%

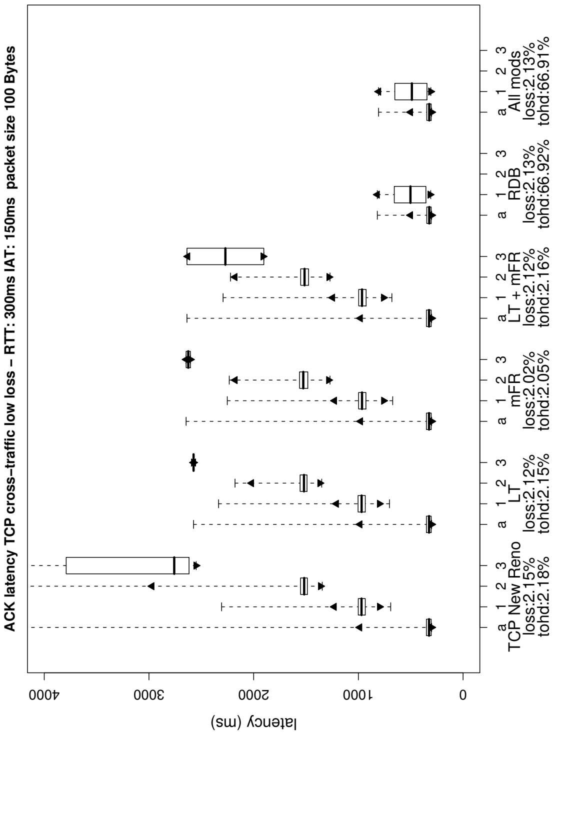

ACK latency TCP cross−traffic low loss − RTT: 250ms IAT: 250ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:3.03%
tohd:4.33%
a 1 2 3
LT
loss:2.87%
tohd:4.09%
a 1 2 3
mFR
loss:3.01%
tohd:4.29%
a 1 2 3
LT + mFR
loss:3.23%
tohd:4.60%
a 1 2 3
RDB
loss:3.26%
tohd:51.92%
a 1 2 3
All mods
loss:3.11%
tohd:51.87%

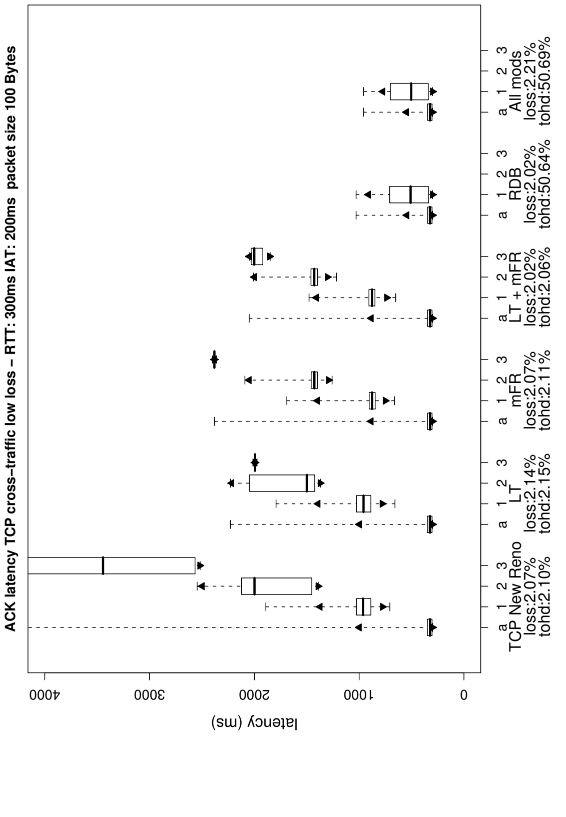

ACK latency TCP cross−traffic low loss − RTT: 250ms IAT: 25ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:2.16%
tohd:2.39%
a 1 2 3
LT
loss:2.16%
tohd:2.41%
a 1 2 3
mFR
loss:2.17%
tohd:2.44%
a 1 2 3
LT + mFR
loss:2.18%
tohd:2.38%
a 1 2 3
RDB
loss:2.76%
tohd:90.49%
a 1 2 3
All mods
loss:2.74%
tohd:90.49%

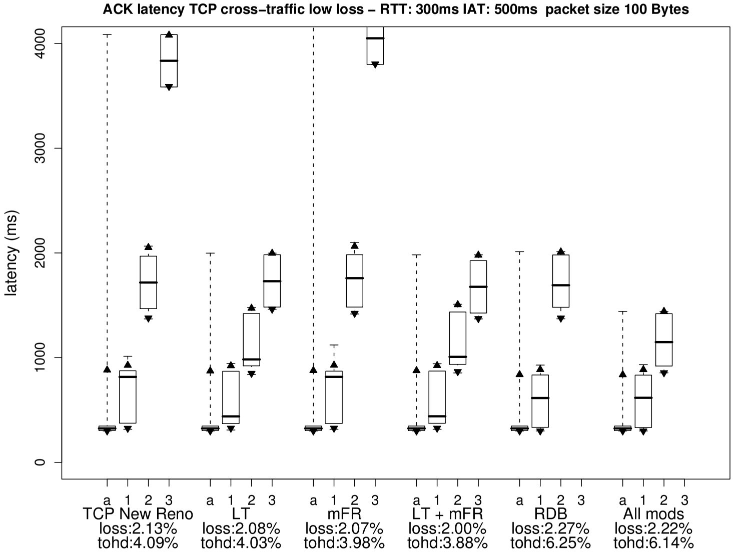

ACK latency TCP cross−traffic low loss − RTT: 250ms IAT: 500ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:2.88%
tohd:4.06%
a 1 2 3
LT
loss:3.09%
tohd:4.38%
a 1 2 3
mFR
loss:2.82%
tohd:3.97%
a 1 2 3
LT + mFR
loss:2.89%
tohd:4.09%
a 1 2 3
RDB
loss:3.17%
tohd:8.51%
a 1 2 3
All mods
loss:2.75%
tohd:7.39%

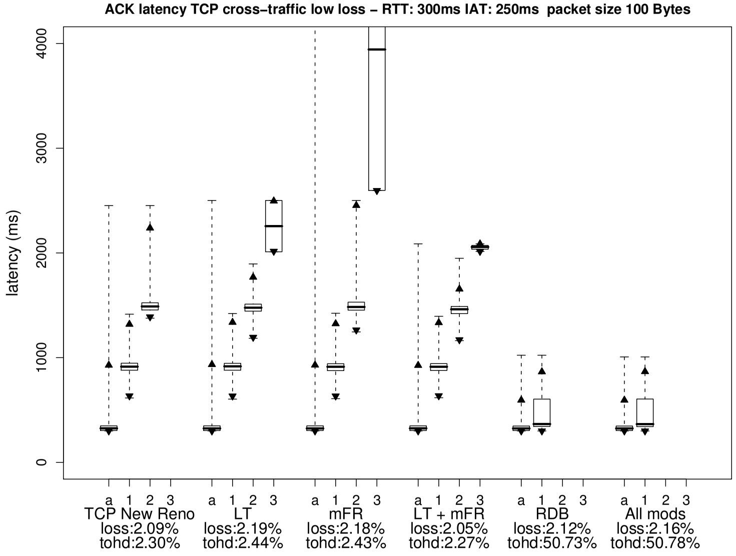

ACK latency TCP cross–traffic low loss – RTT: 250ms IAT: 50ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:2.57%
tohd:2.65%
a 1 2 3
LT
loss:2.49%
tohd:2.54%
a 1 2 3
mFR
loss:2.51%
tohd:2.61%
a 1 2 3
LT + mFR
loss:2.51%
tohd:2.62%
a 1 2 3
RDB
loss:2.80%
tohd:84.17%
a 1 2 3
All mods
loss:2.85%
tohd:84.17%

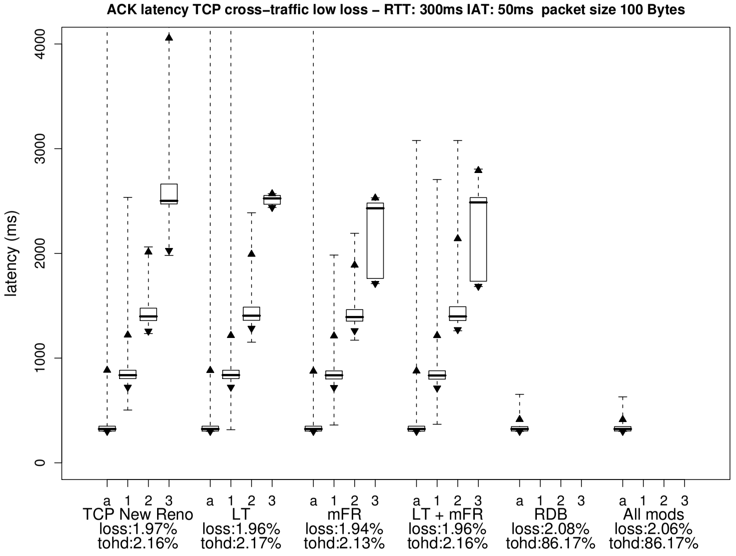

ACK latency TCP cross–traffic low loss – RTT: 300ms IAT: 100ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:2.11%
tohd:2.18%
a 1 2 3
LT
loss:2.09%
tohd:2.18%
a 1 2 3
mFR
loss:2.06%
tohd:2.15%
a 1 2 3
LT + mFR
loss:2.08%
tohd:2.12%
a 1 2 3
RDB
loss:2.03%
tohd:75.13%
a 1 2 3
All mods
loss:2.18%
tohd:75.14%

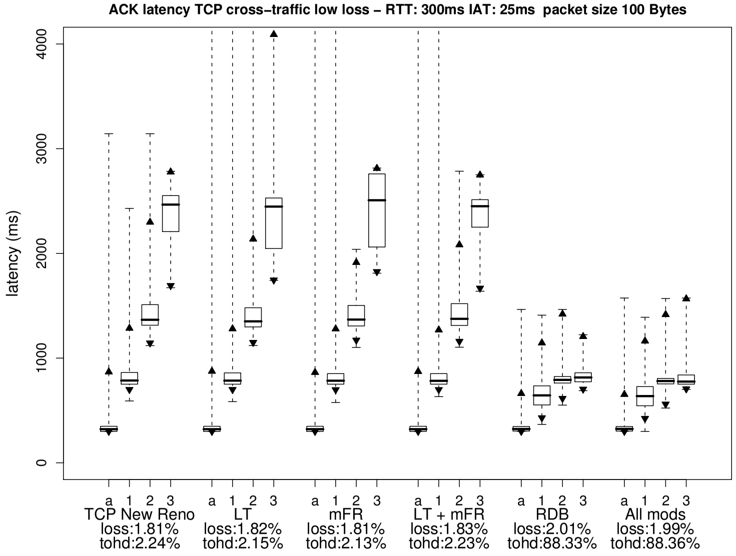

ACK latency TCP cross–traffic low loss – RTT: 300ms IAT: 150ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:2.15%
tohd:2.18%
a 1 2 3
LT
loss:2.12%
tohd:2.15%
a 1 2 3
mFR
loss:2.02%
tohd:2.05%
a 1 2 3
LT + mFR
loss:2.12%
tohd:2.16%
a 1 2 3
RDB
loss:2.13%
tohd:66.92%
a 1 2 3
All mods
loss:2.13%
tohd:66.91%

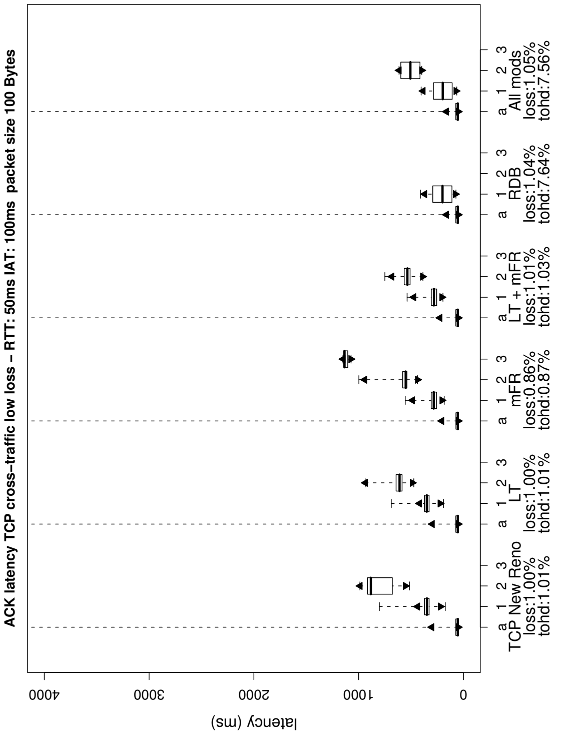

ACK latency TCP cross–traffic low loss – RTT: 300ms IAT: 200ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:2.07%
tohd:2.10%
a 1 2 3
LT
loss:2.14%
tohd:2.15%
a 1 2 3
mFR
loss:2.07%
tohd:2.11%
a 1 2 3
LT + mFR
loss:2.02%
tohd:2.06%
a 1 2 3
RDB
loss:2.02%
tohd:50.64%
a 1 2 3
All mods
loss:2.21%
tohd:50.69%

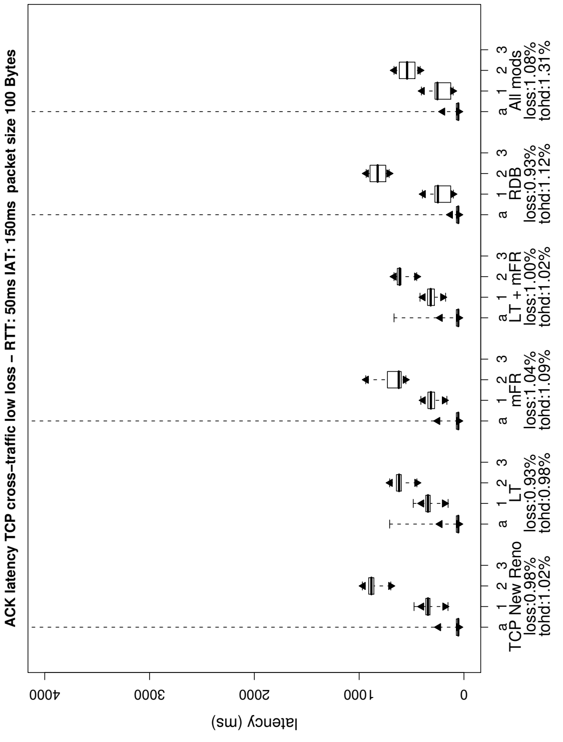

ACK latency TCP cross−traffic low loss − RTT: 300ms IAT: 500ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:2.13%
tohd:4.09%
a 1 2 3
LT
loss:2.08%
tohd:4.03%
a 1 2 3
mFR
loss:2.07%
tohd:3.98%
a 1 2 3
LT + mFR
loss:2.00%
tohd:3.88%
a 1 2 3
RDB
loss:2.27%
tohd:6.25%
a 1 2 3
All mods
loss:2.22%
tohd:6.14%

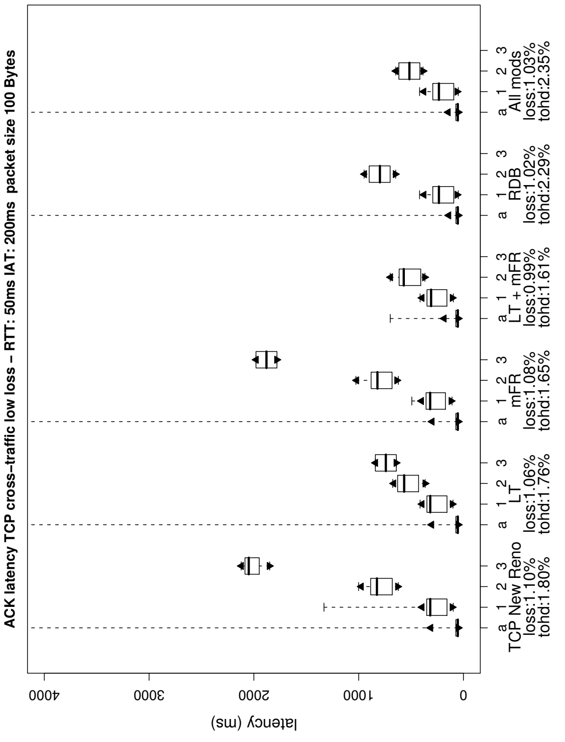

ACK latency TCP cross−traffic low loss − RTT: 300ms IAT: 250ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:2.09%
tohd:2.30%
a 1 2 3
LT
loss:2.19%
tohd:2.44%
a 1 2 3
mFR
loss:2.18%
tohd:2.43%
a 1 2 3
LT + mFR
loss:2.05%
tohd:2.27%
a 1 2 3
RDB
loss:2.12%
tohd:50.73%
a 1 2 3
All mods
loss:2.16%
tohd:50.78%

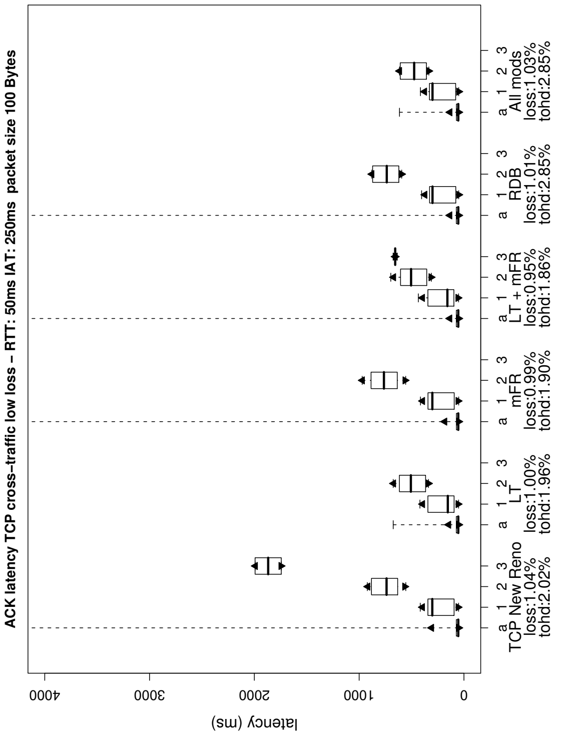

ACK latency TCP cross−traffic low loss − RTT: 300ms IAT: 50ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:1.97%
tohd:2.16%
a 1 2 3
LT
loss:1.96%
tohd:2.17%
a 1 2 3
mFR
loss:1.94%
tohd:2.13%
a 1 2 3
LT + mFR
loss:1.96%
tohd:2.16%
a 1 2 3
RDB
loss:2.08%
tohd:86.17%
a 1 2 3
All mods
loss:2.06%
tohd:86.17%

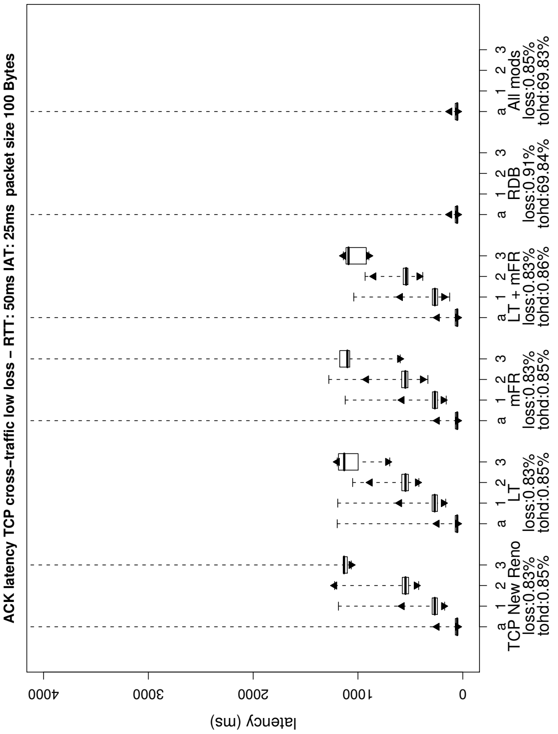

ACK latency TCP cross−traffic low loss − RTT: 300ms IAT: 25ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:1.81%
tohd:2.24%
a 1 2 3
LT
loss:1.82%
tohd:2.15%
a 1 2 3
mFR
loss:1.81%
tohd:2.13%
a 1 2 3
LT + mFR
loss:1.83%
tohd:2.23%
a 1 2 3
RDB
loss:2.01%
tohd:88.33%
a 1 2 3
All mods
loss:1.99%
tohd:88.36%

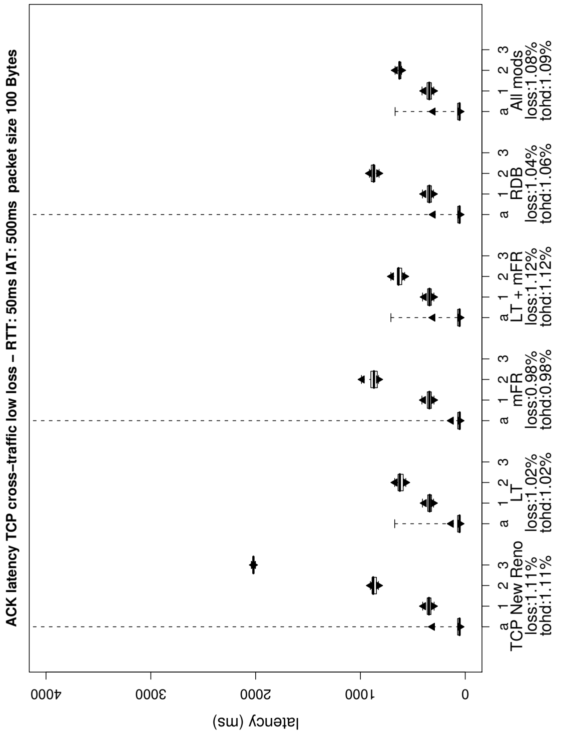

ACK latency TCP cross−traffic low loss − RTT: 50ms IAT: 100ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:1.00%
tohd:1.01%
a 1 2 3
LT
loss:1.00%
tohd:1.01%
a 1 2 3
mFR
loss:0.86%
tohd:0.87%
a 1 2 3
LT + mFR
loss:1.01%
tohd:1.03%
a 1 2 3
RDB
loss:1.04%
tohd:7.64%
a 1 2 3
All mods
loss:1.05%
tohd:7.56%

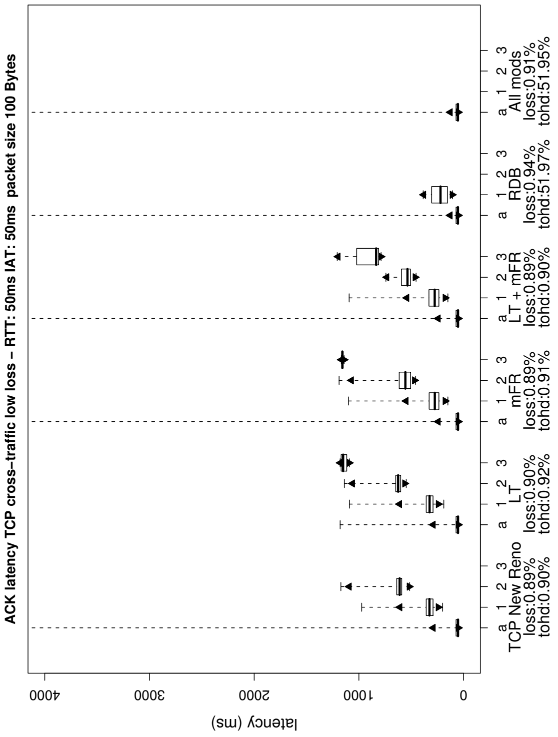

ACK latency TCP cross−traffic low loss − RTT: 50ms IAT: 150ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:0.98%
tohd:1.02%
a 1 2 3
LT
loss:0.93%
tohd:0.98%
a 1 2 3
mFR
loss:1.04%
tohd:1.09%
a 1 2 3
LT + mFR
loss:1.00%
tohd:1.02%
a 1 2 3
RDB
loss:0.93%
tohd:1.12%
a 1 2 3
All mods
loss:1.08%
tohd:1.31%


ACK latency TCP cross−traffic low loss − RTT: 50ms IAT: 200ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:1.10%
tohd:1.80%
a 1 2 3
LT
loss:1.06%
tohd:1.76%
a 1 2 3
mFR
loss:1.08%
tohd:1.65%
a 1 2 3
LT + mFR
loss:0.99%
tohd:1.61%
a 1 2 3
RDB
loss:1.02%
tohd:2.29%
a 1 2 3
All mods
loss:1.03%
tohd:2.35%


ACK latency TCP cross−traffic low loss − RTT: 50ms IAT: 250ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:1.04%
tohd:2.02%
a 1 2 3
LT
loss:1.00%
tohd:1.96%
a 1 2 3
mFR
loss:0.99%
tohd:1.90%
a 1 2 3
LT + mFR
loss:0.95%
tohd:1.86%
a 1 2 3
RDB
loss:1.01%
tohd:2.85%
a 1 2 3
All mods
loss:1.03%
tohd:2.85%


ACK latency TCP cross−traffic low loss − RTT: 50ms IAT: 25ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:0.83%
tohd:0.85%
a 1 2 3
LT
loss:0.83%
tohd:0.85%
a 1 2 3
mFR
loss:0.83%
tohd:0.85%
a 1 2 3
LT + mFR
loss:0.83%
tohd:0.86%
a 1 2 3
RDB
loss:0.91%
tohd:69.84%
a 1 2 3
All mods
loss:0.85%
tohd:69.83%


ACK latency TCP cross−traffic low loss − RTT: 50ms IAT: 500ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:1.11%
tohd:1.11%
a 1 2 3
LT
loss:1.02%
tohd:1.02%
a 1 2 3
mFR
loss:0.98%
tohd:0.98%
a 1 2 3
LT + mFR
loss:1.12%
tohd:1.12%
a 1 2 3
RDB
loss:1.04%
tohd:1.06%
a 1 2 3
All mods
loss:1.08%
tohd:1.09%


ACK latency TCP cross−traffic low loss − RTT: 50ms IAT: 50ms packet size 100 Bytes
latency (ms)
0
1000
2000
3000
4000
a 1 2 3
TCP New Reno
loss:0.89%
tohd:0.90%
a 1 2 3
LT
loss:0.90%
tohd:0.92%
a 1 2 3
mFR
loss:0.89%
tohd:0.91%
a 1 2 3
LT + mFR
loss:0.89%
tohd:0.90%
a 1 2 3
RDB
loss:0.94%
tohd:51.97%
a 1 2 3
All mods
loss:0.91%
tohd:51.95%

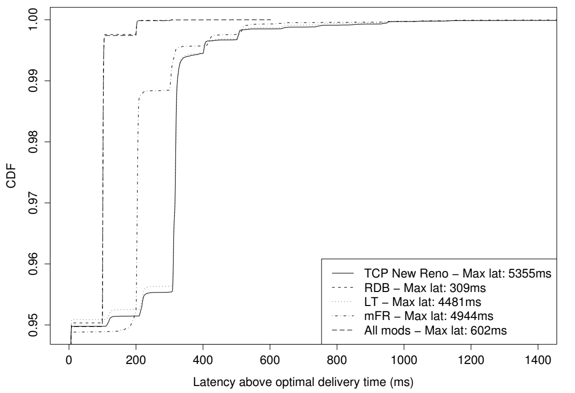
Appendix D

Complete set of CDFs from TCP laboratory experiments

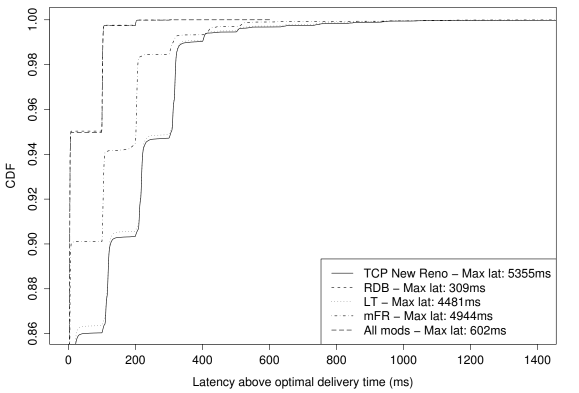
This appendix contains the whole range of CDFs from our laboratory experiments using the thin-stream TCP modifications. Transport layer delivery latency and application layer delivery latency are presented pairwise for each respective test. This is an extension of the selected statistics that are presented and discussed in section 5.2.

D.1 Uniform loss

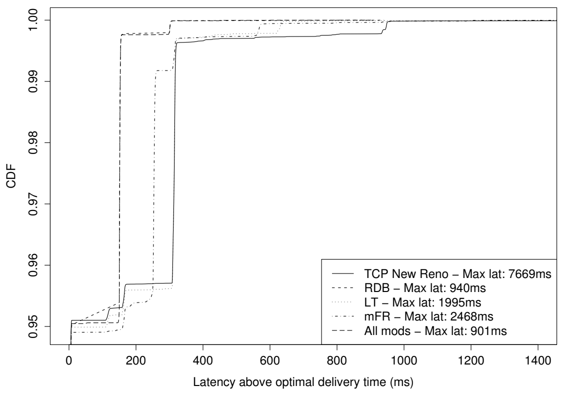
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 100ms IAT: 100ms packet size 100 Bytes



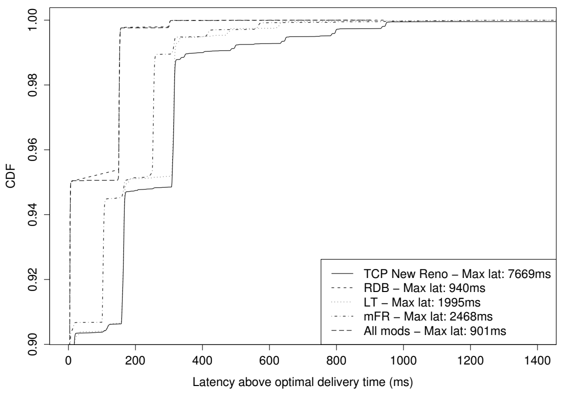
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 100ms IAT: 100ms packet size 100 Bytes



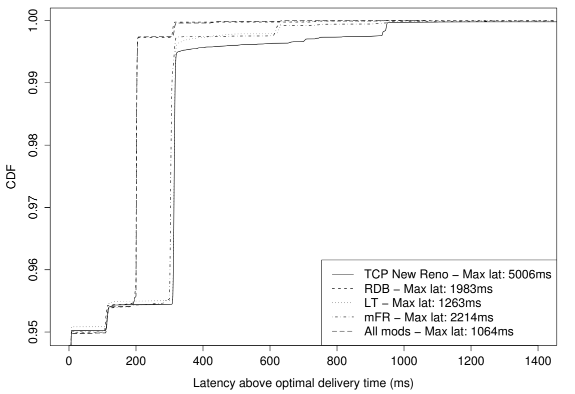
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 100ms IAT: 150ms packet size 100 Bytes



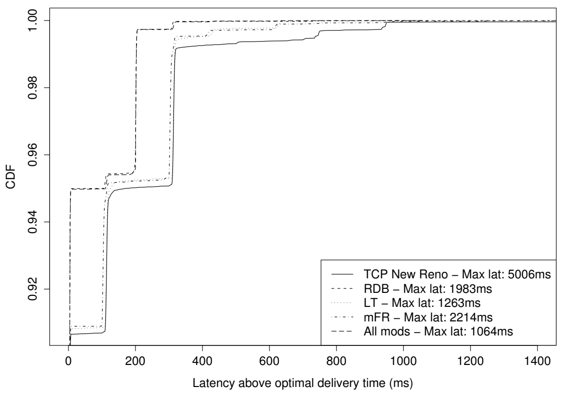
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 100ms IAT: 150ms packet size 100 Bytes



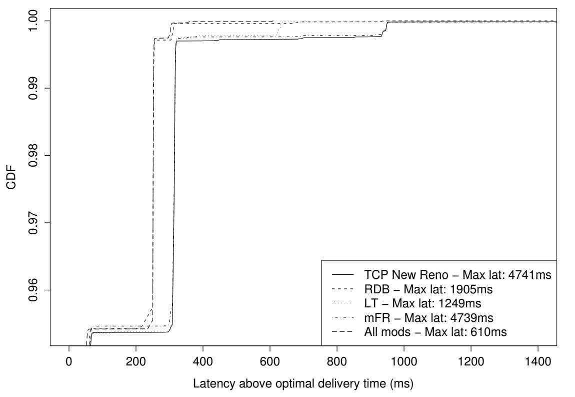
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 100ms IAT: 200ms packet size 100 Bytes



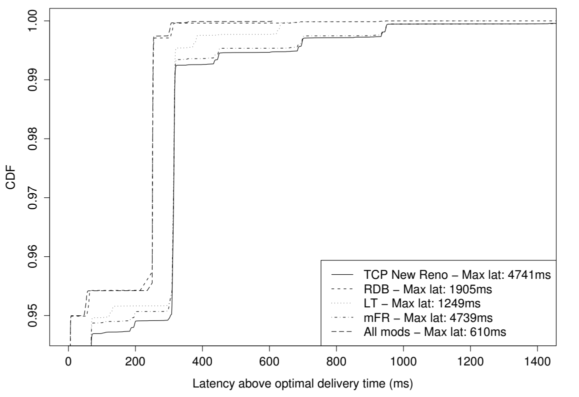
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 100ms IAT: 200ms packet size 100 Bytes



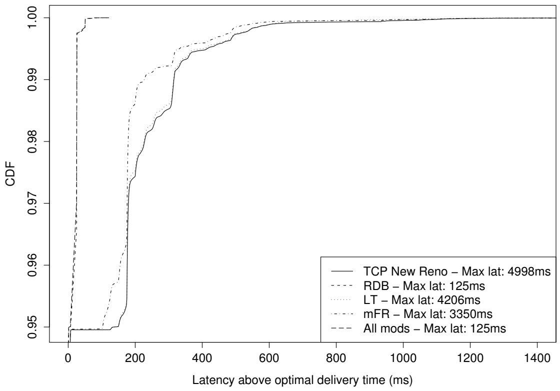
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 100ms IAT: 250ms packet size 100 Bytes



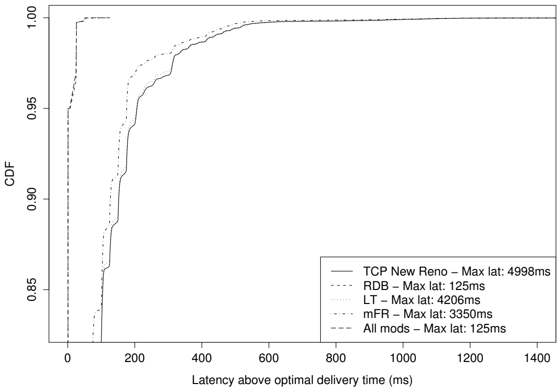
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 100ms IAT: 250ms packet size 100 Bytes



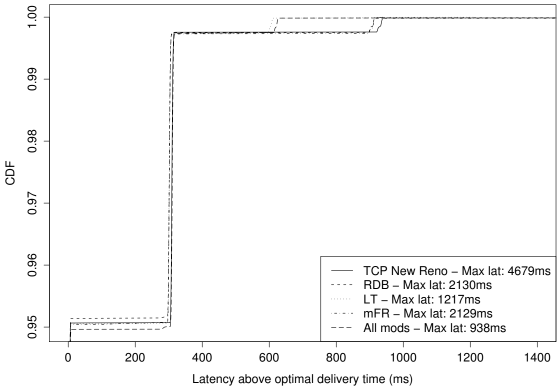
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 100ms IAT: 25ms packet size 100 Bytes



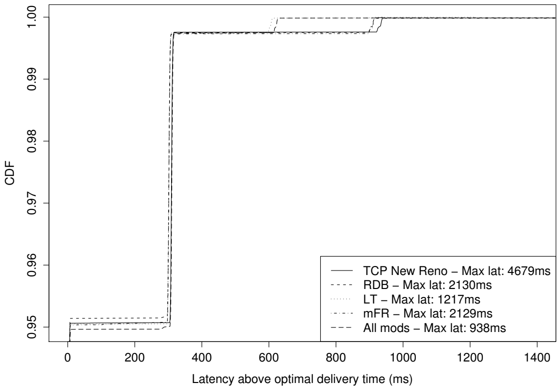
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 100ms IAT: 25ms packet size 100 Bytes



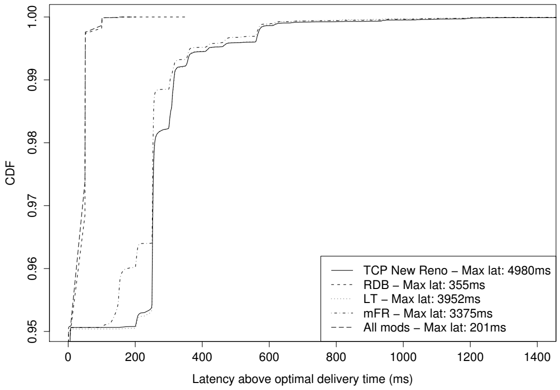
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 100ms IAT: 500ms packet size 100 Bytes



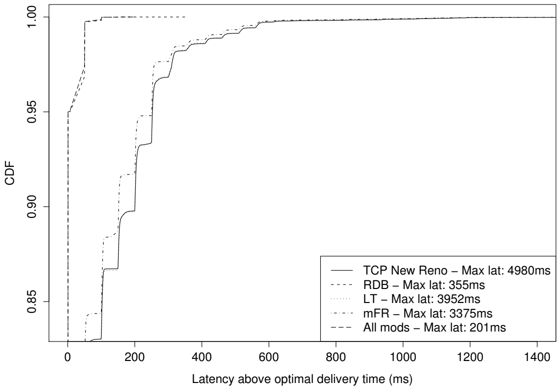
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 100ms IAT: 500ms packet size 100 Bytes



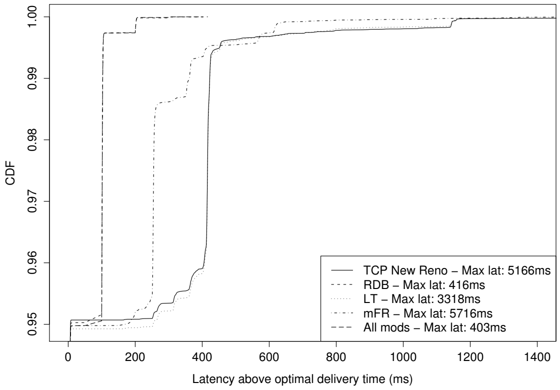
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 100ms IAT: 50ms packet size 100 Bytes



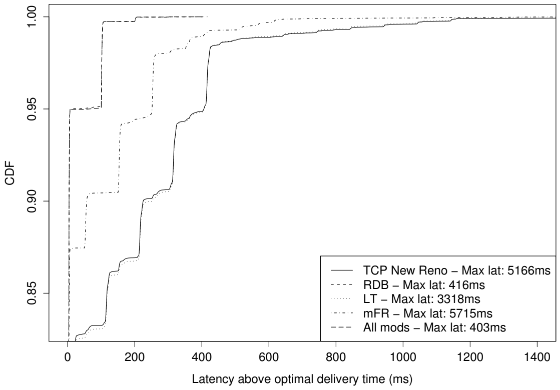
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 100ms IAT: 50ms packet size 100 Bytes

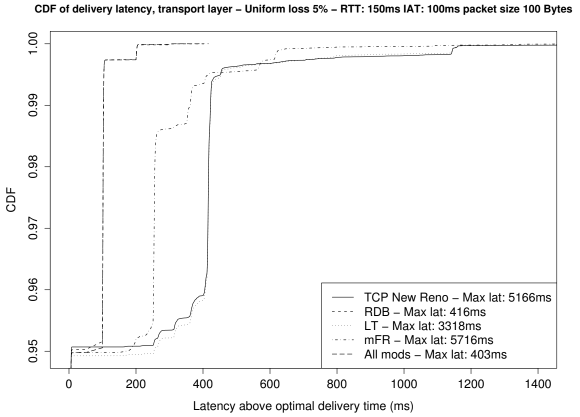


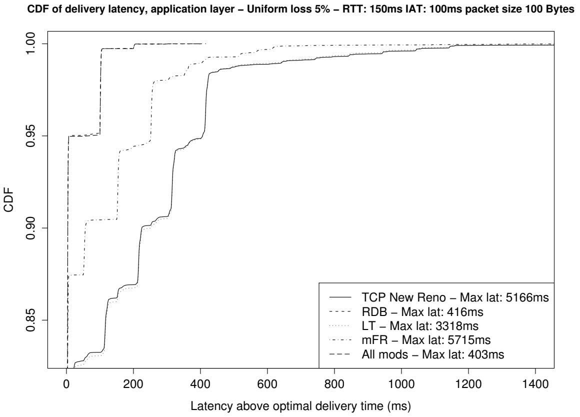
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 150ms IAT: 100ms packet size 100 Bytes

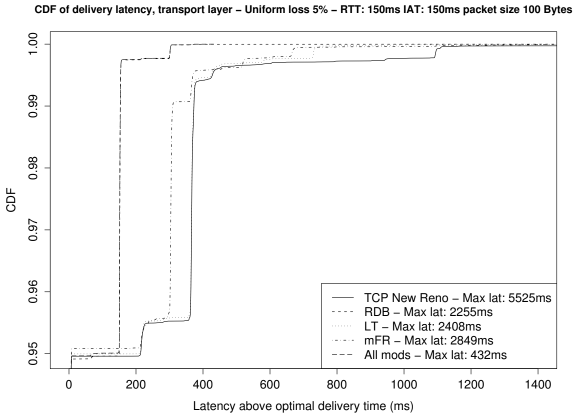


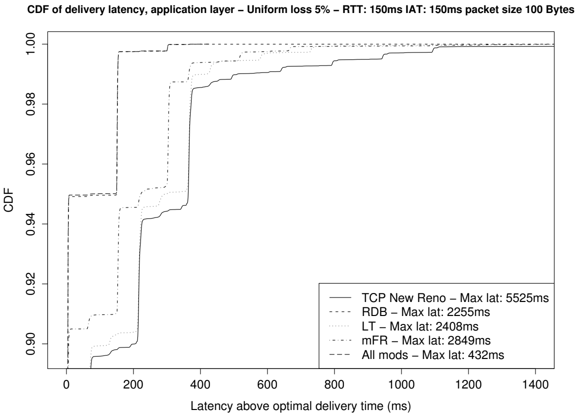
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 150ms IAT: 100ms packet size 100 Bytes

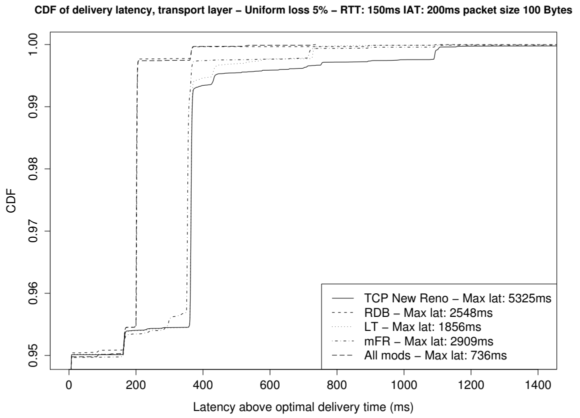


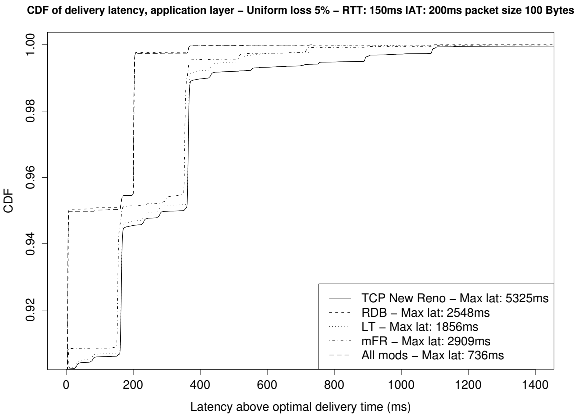

CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 150ms IAT: 100ms packet size 100 Bytes

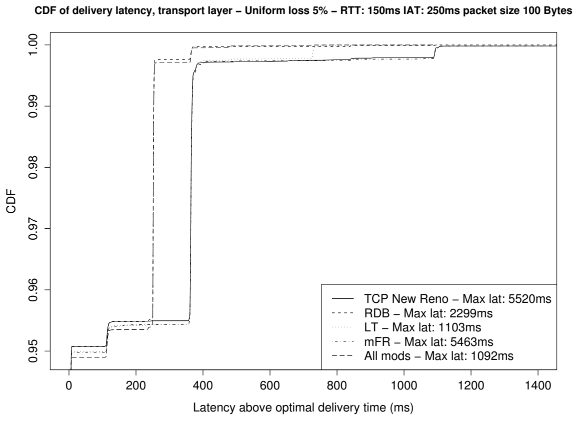

CDF of delivery latency, application layer – Uniform loss 5% – RTT: 150ms IAT: 100ms packet size 100 Bytes

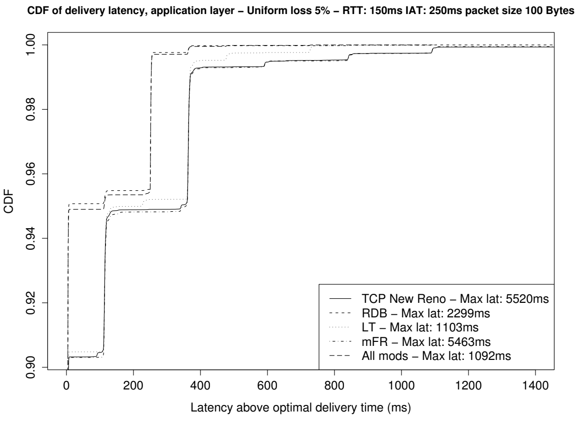

CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 150ms IAT: 150ms packet size 100 Bytes

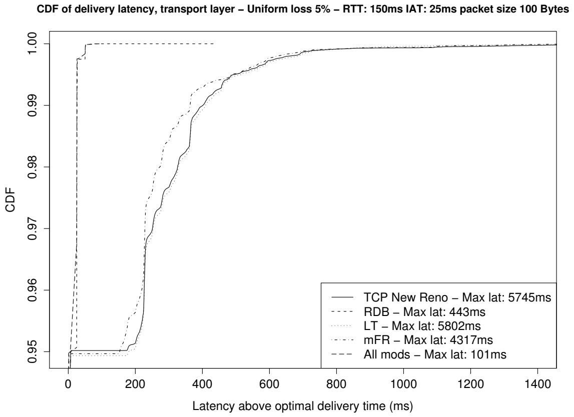

CDF of delivery latency, application layer – Uniform loss 5% – RTT: 150ms IAT: 150ms packet size 100 Bytes

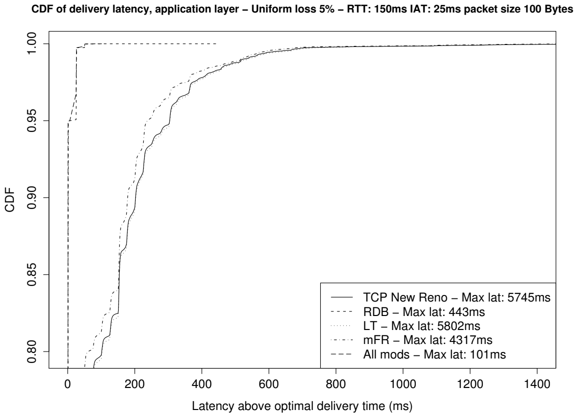

CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 150ms IAT: 200ms packet size 100 Bytes

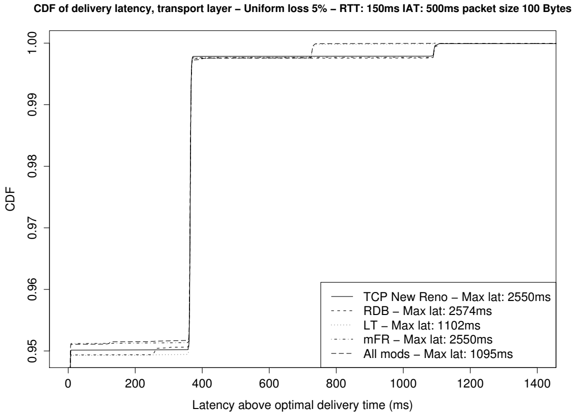

CDF of delivery latency, application layer – Uniform loss 5% – RTT: 150ms IAT: 200ms packet size 100 Bytes

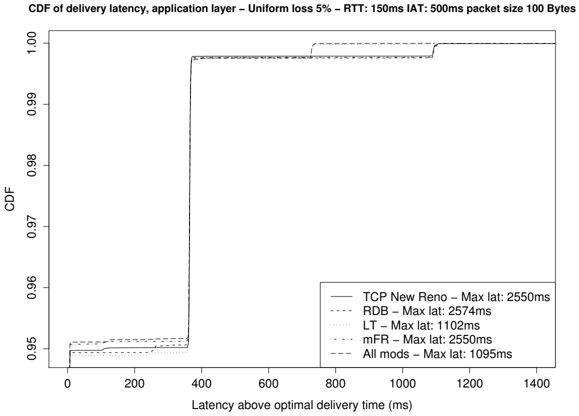

CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 150ms IAT: 250ms packet size 100 Bytes

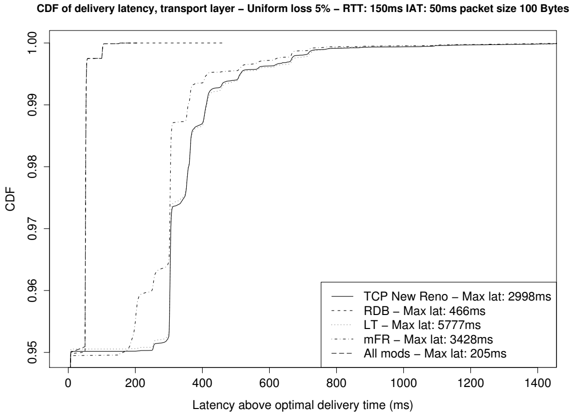

CDF of delivery latency, application layer – Uniform loss 5% – RTT: 150ms IAT: 250ms packet size 100 Bytes

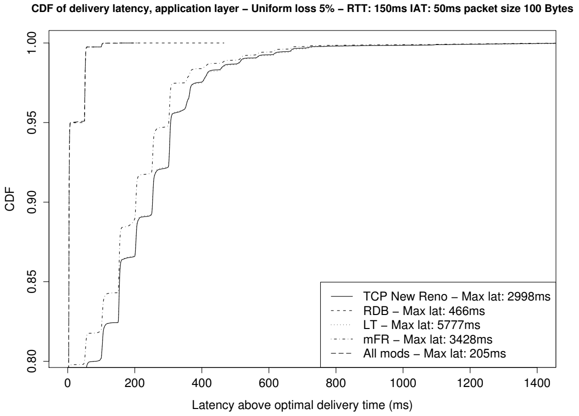

CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 150ms IAT: 25ms packet size 100 Bytes
CDF
0.95
0.96
0.97
0.98
0.99
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 5745ms
RDB – Max lat: 443ms
LT – Max lat: 5802ms
mFR – Max lat: 4317ms
All mods – Max lat: 101ms

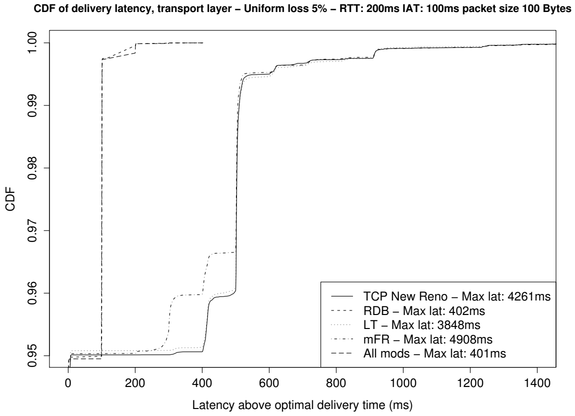

CDF of delivery latency, application layer – Uniform loss 5% – RTT: 150ms IAT: 25ms packet size 100 Bytes
CDF
0.80
0.85
0.90
0.95
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 5745ms
RDB – Max lat: 443ms
LT – Max lat: 5802ms
mFR – Max lat: 4317ms
All mods – Max lat: 101ms

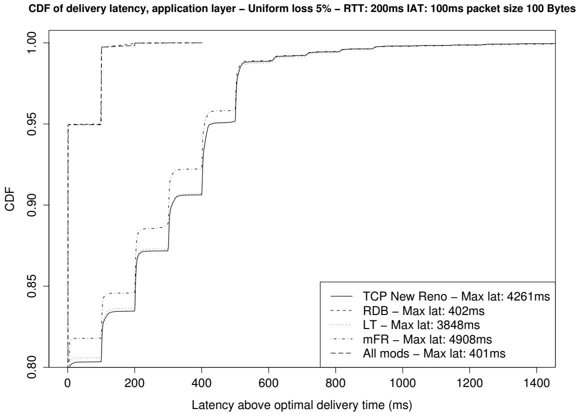

CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 150ms IAT: 500ms packet size 100 Bytes
CDF
0.95
0.96
0.97
0.98
0.99
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 2550ms
RDB – Max lat: 2574ms
LT – Max lat: 1102ms
mFR – Max lat: 2550ms
All mods – Max lat: 1095ms

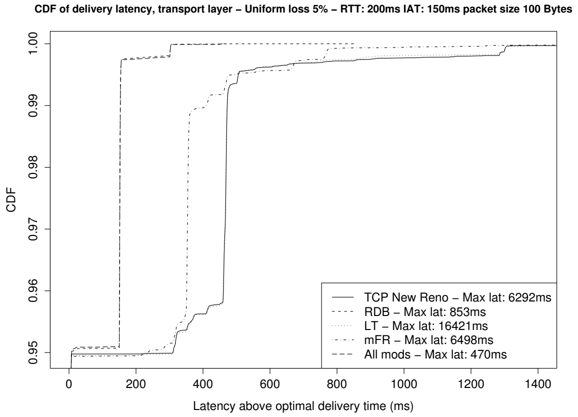

CDF of delivery latency, application layer – Uniform loss 5% – RTT: 150ms IAT: 500ms packet size 100 Bytes
CDF
0.95
0.96
0.97
0.98
0.99
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 2550ms
RDB – Max lat: 2574ms
LT – Max lat: 1102ms
mFR – Max lat: 2550ms
All mods – Max lat: 1095ms

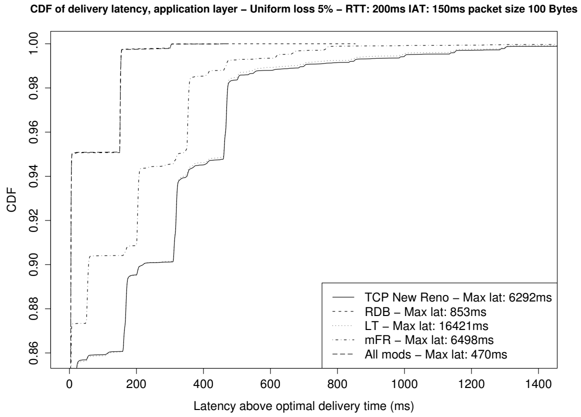

CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 150ms IAT: 50ms packet size 100 Bytes
CDF
0.95
0.96
0.97
0.98
0.99
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 2998ms
RDB – Max lat: 466ms
LT – Max lat: 5777ms
mFR – Max lat: 3428ms
All mods – Max lat: 205ms

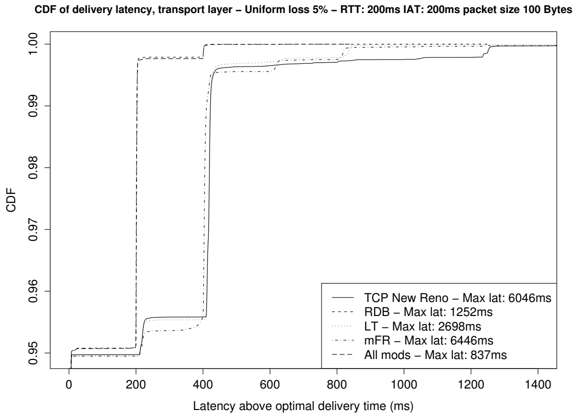

CDF of delivery latency, application layer – Uniform loss 5% – RTT: 150ms IAT: 50ms packet size 100 Bytes
CDF
0.80
0.85
0.90
0.95
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 2998ms
RDB – Max lat: 466ms
LT – Max lat: 5777ms
mFR – Max lat: 3428ms
All mods – Max lat: 205ms

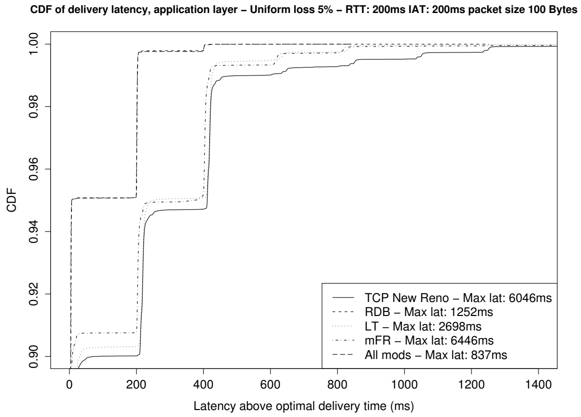

CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 200ms IAT: 100ms packet size 100 Bytes
CDF
0.95
0.96
0.97
0.98
0.99
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 4261ms
RDB – Max lat: 402ms
LT – Max lat: 3848ms
mFR – Max lat: 4908ms
All mods – Max lat: 401ms

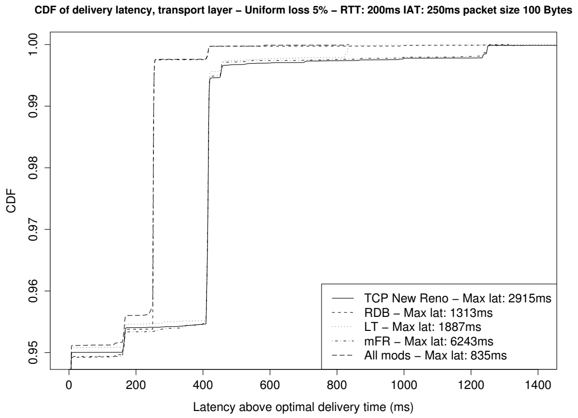

CDF of delivery latency, application layer – Uniform loss 5% – RTT: 200ms IAT: 100ms packet size 100 Bytes
CDF
0.80
0.85
0.90
0.95
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 4261ms
RDB – Max lat: 402ms
LT – Max lat: 3848ms
mFR – Max lat: 4908ms
All mods – Max lat: 401ms

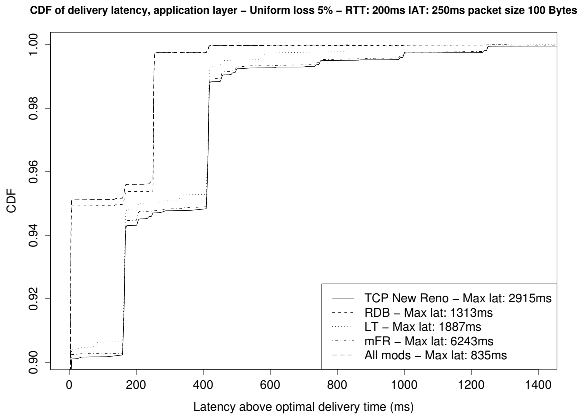

CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 200ms IAT: 150ms packet size 100 Bytes

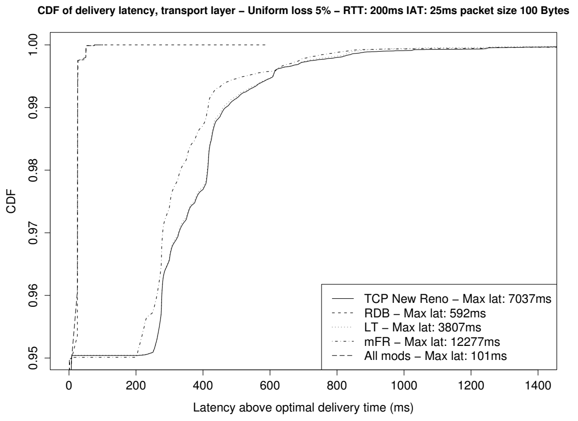

CDF of delivery latency, application layer – Uniform loss 5% – RTT: 200ms IAT: 150ms packet size 100 Bytes

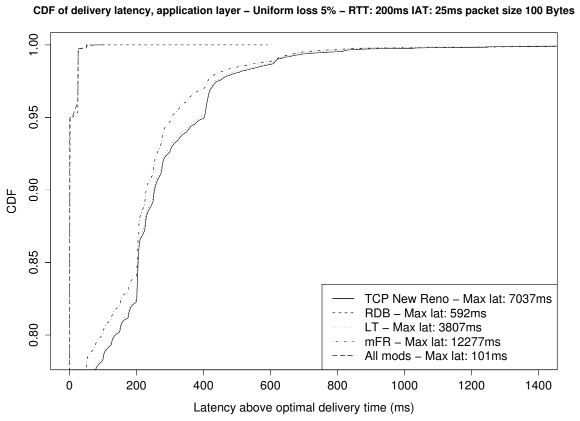

CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 200ms IAT: 200ms packet size 100 Bytes


CDF of delivery latency, application layer – Uniform loss 5% – RTT: 200ms IAT: 200ms packet size 100 Bytes

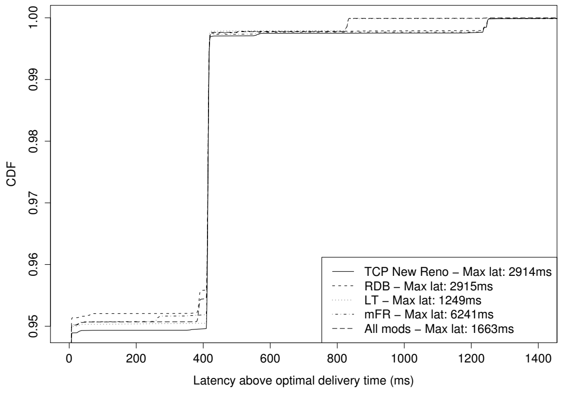

CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 200ms IAT: 250ms packet size 100 Bytes


CDF of delivery latency, application layer – Uniform loss 5% – RTT: 200ms IAT: 250ms packet size 100 Bytes

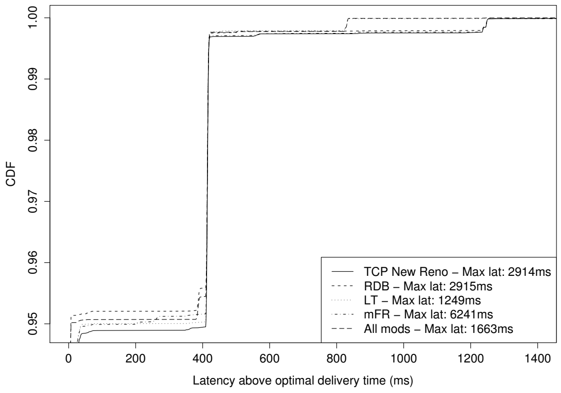

CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 200ms IAT: 25ms packet size 100 Bytes


CDF of delivery latency, application layer – Uniform loss 5% – RTT: 200ms IAT: 25ms packet size 100 Bytes

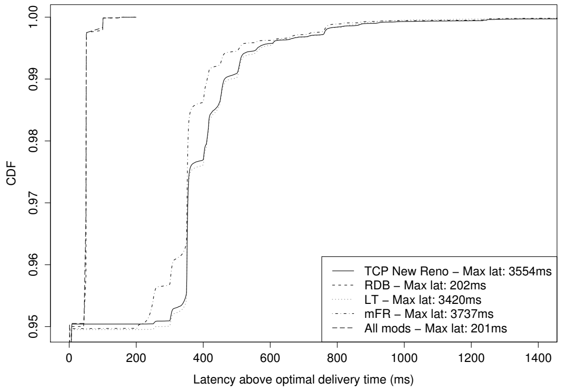
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 200ms IAT: 500ms packet size 100 Bytes



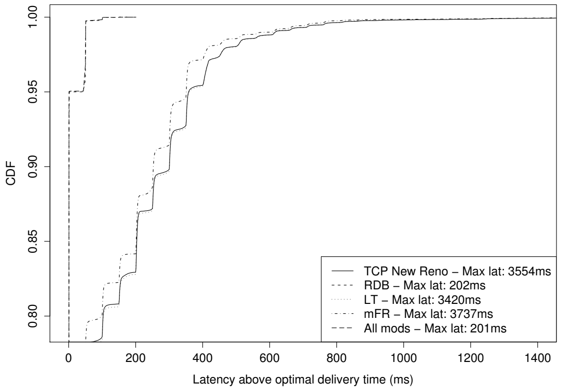
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 200ms IAT: 500ms packet size 100 Bytes



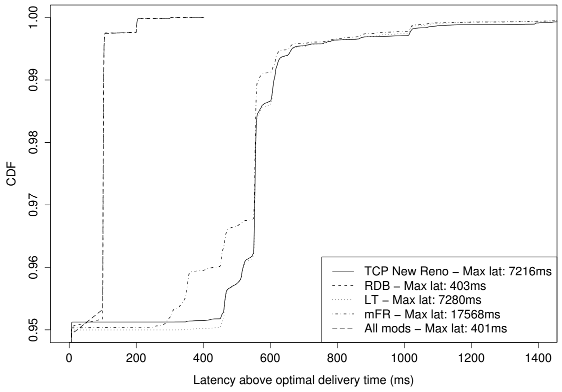
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 200ms IAT: 50ms packet size 100 Bytes



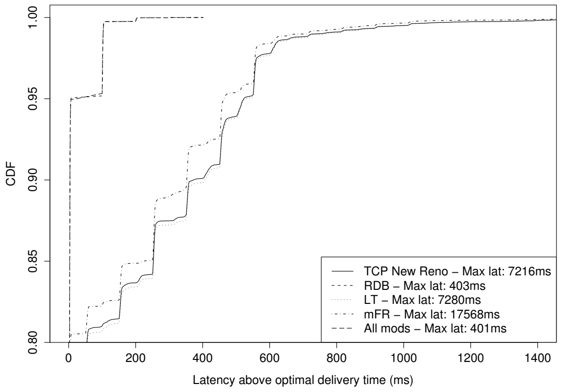
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 200ms IAT: 50ms packet size 100 Bytes



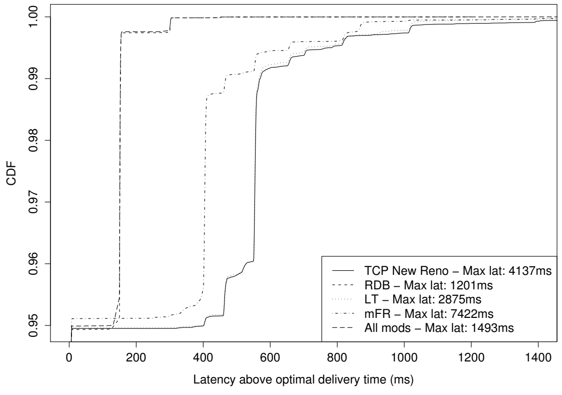
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 250ms IAT: 100ms packet size 100 Bytes



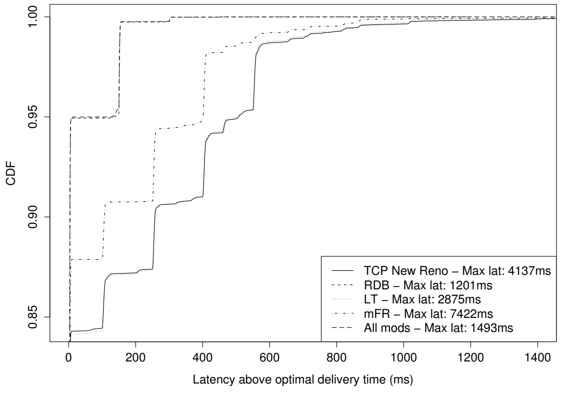
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 250ms IAT: 100ms packet size 100 Bytes



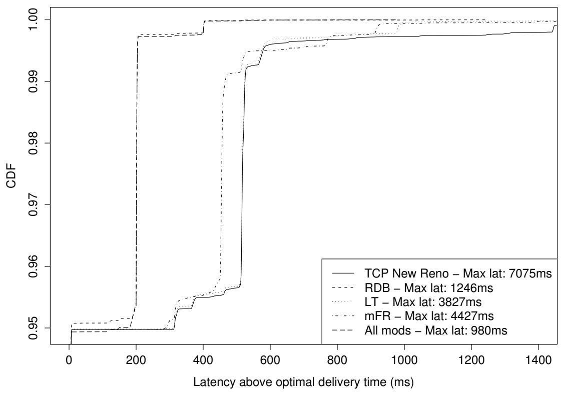
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 250ms IAT: 150ms packet size 100 Bytes



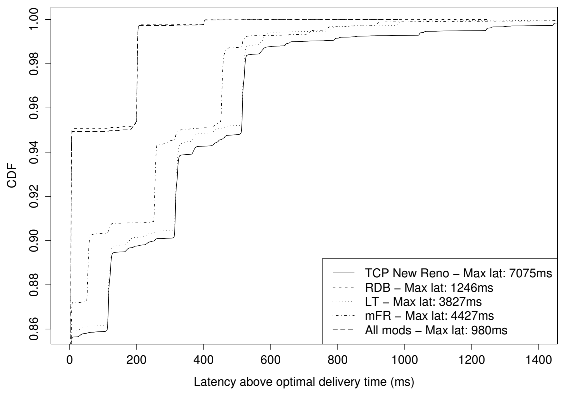
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 250ms IAT: 150ms packet size 100 Bytes



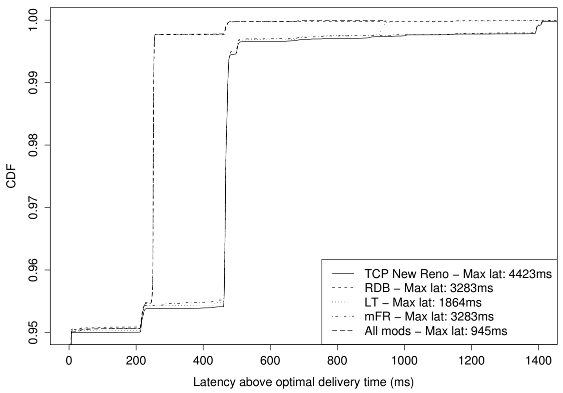
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 250ms IAT: 200ms packet size 100 Bytes



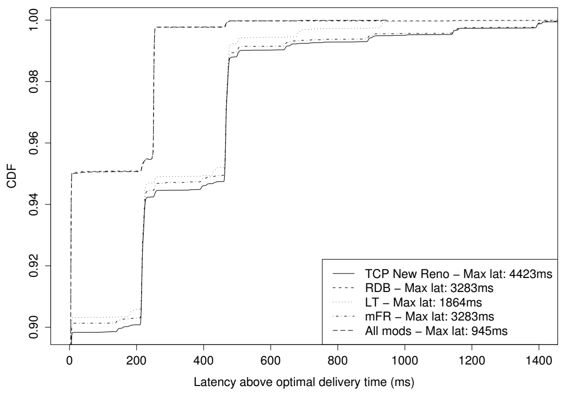
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 250ms IAT: 200ms packet size 100 Bytes



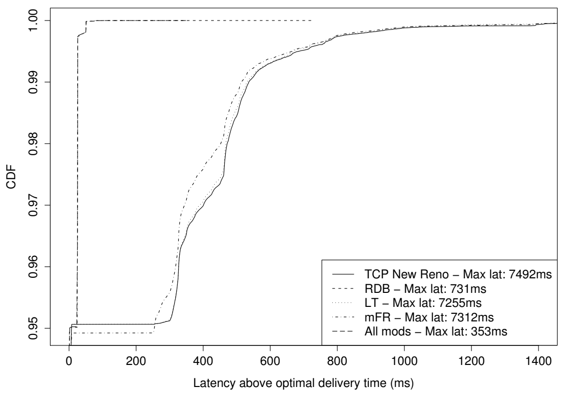
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 250ms IAT: 250ms packet size 100 Bytes



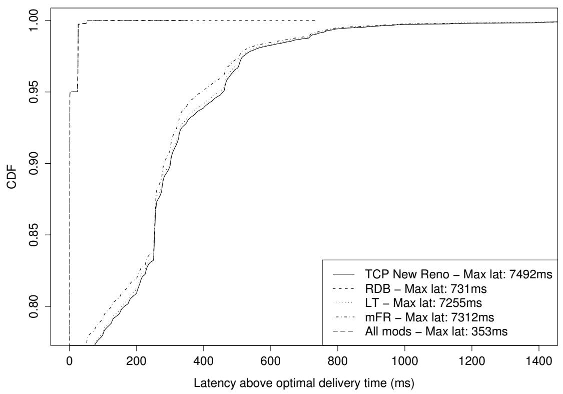
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 250ms IAT: 250ms packet size 100 Bytes



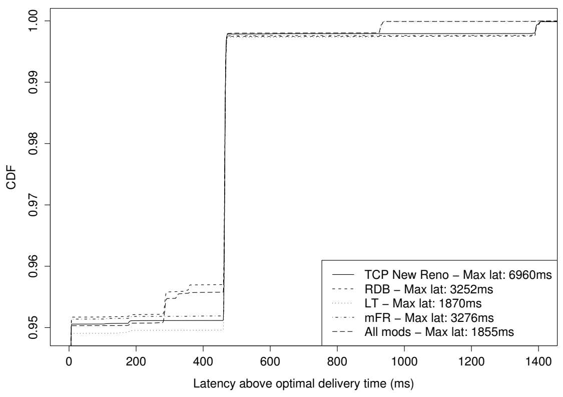
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 250ms IAT: 25ms packet size 100 Bytes



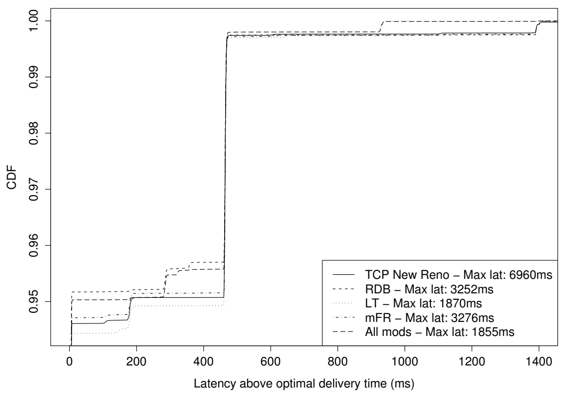
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 250ms IAT: 25ms packet size 100 Bytes



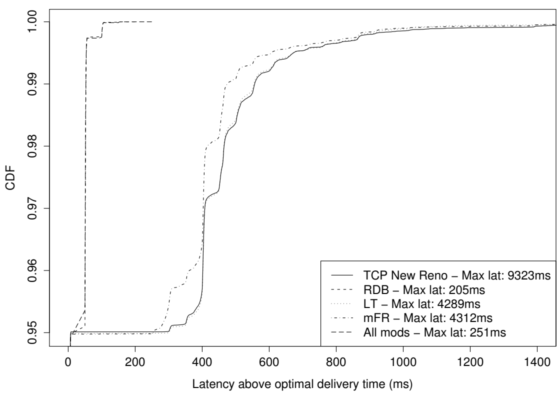
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 250ms IAT: 500ms packet size 100 Bytes



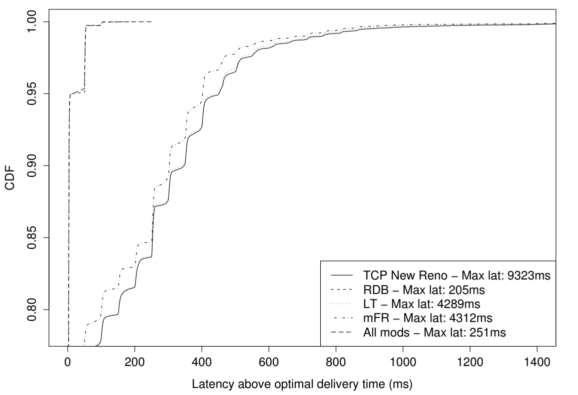
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 250ms IAT: 500ms packet size 100 Bytes



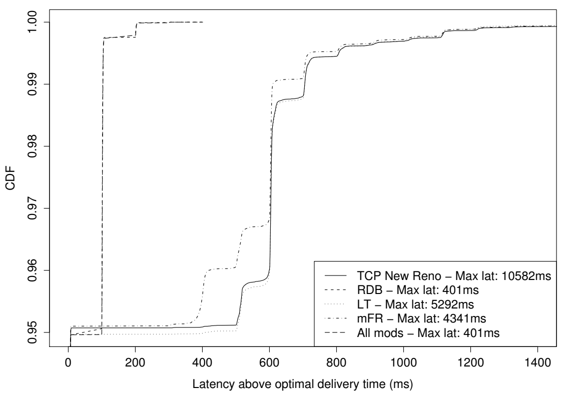
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 250ms IAT: 50ms packet size 100 Bytes



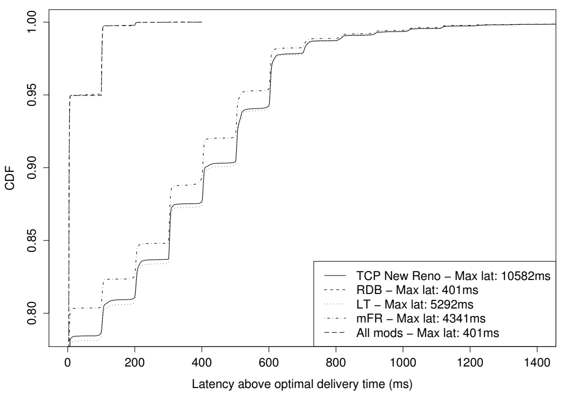
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 250ms IAT: 50ms packet size 100 Bytes



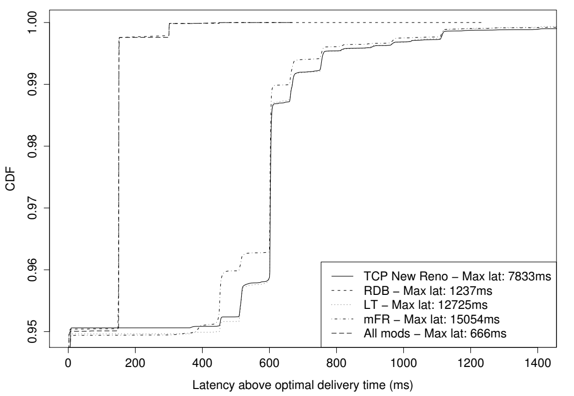
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 300ms IAT: 100ms packet size 100 Bytes



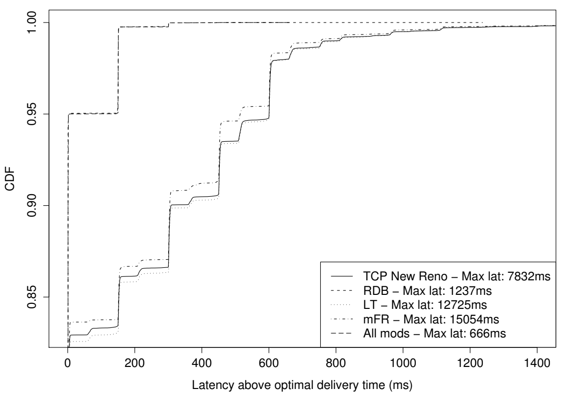
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 300ms IAT: 100ms packet size 100 Bytes



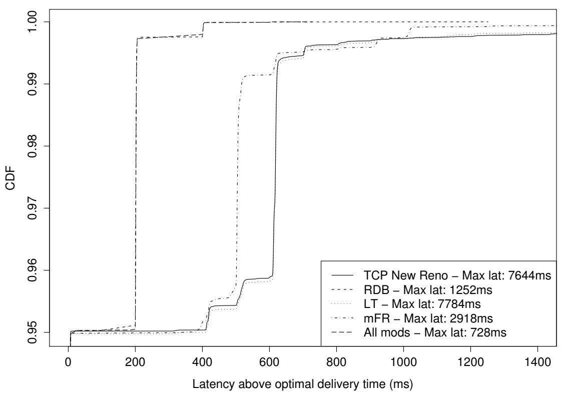
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 300ms IAT: 150ms packet size 100 Bytes



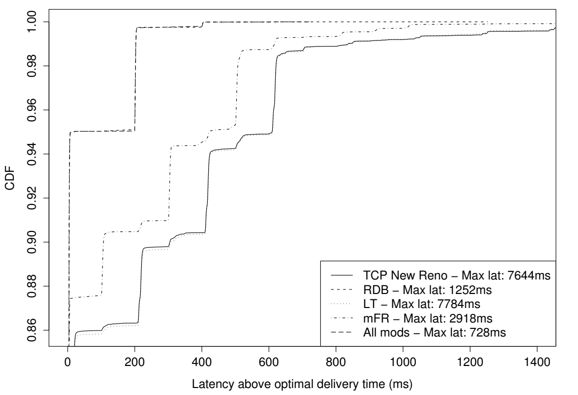
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 300ms IAT: 150ms packet size 100 Bytes

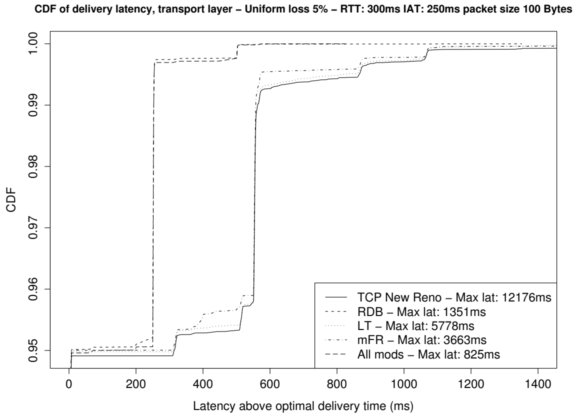


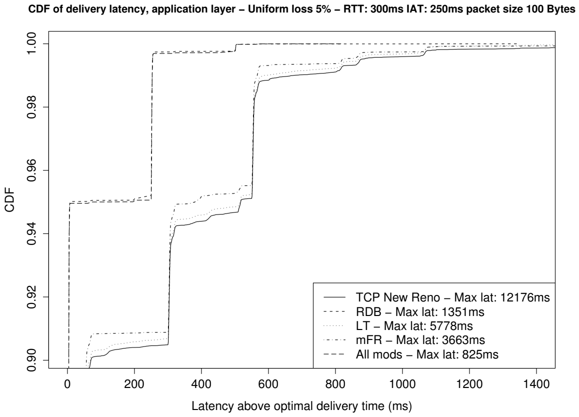
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 300ms IAT: 200ms packet size 100 Bytes

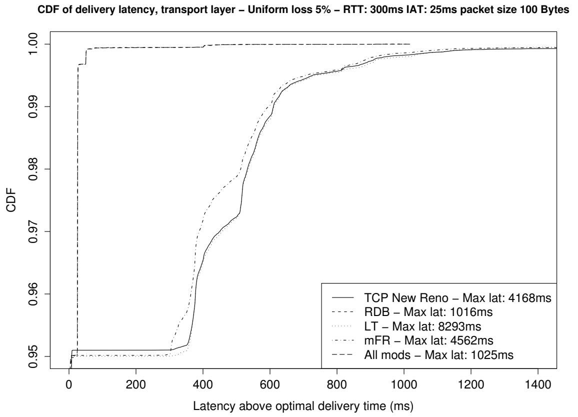


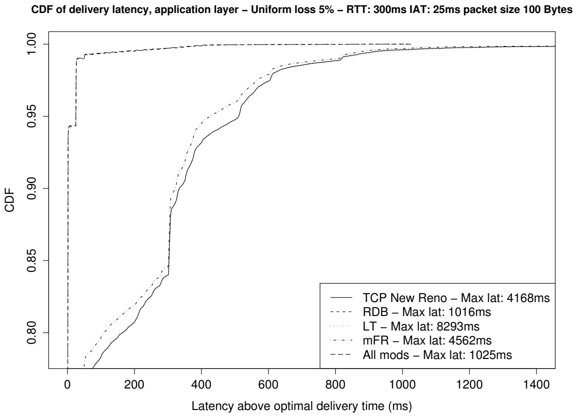
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 300ms IAT: 200ms packet size 100 Bytes

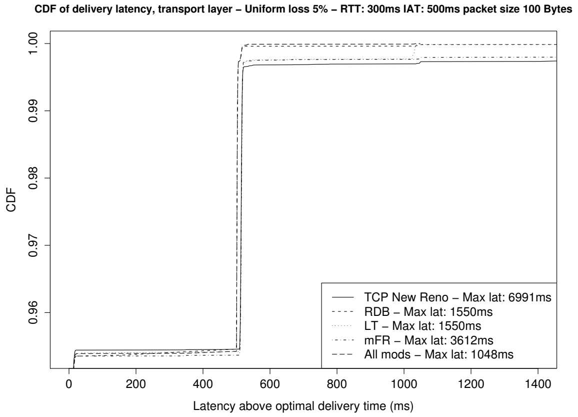


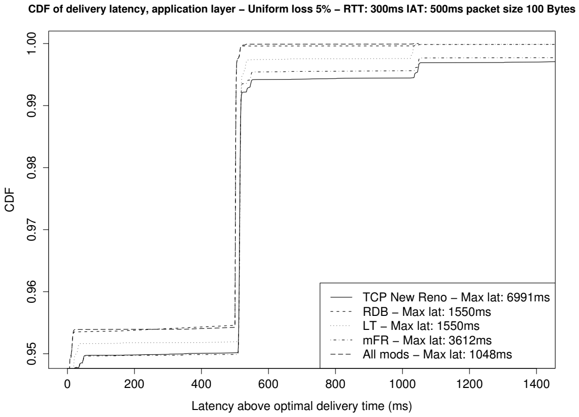

CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 300ms IAT: 250ms packet size 100 Bytes
CDF
0.95
0.96
0.97
0.98
0.99
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 12176ms
RDB – Max lat: 1351ms
LT – Max lat: 5778ms
mFR – Max lat: 3663ms
All mods – Max lat: 825ms

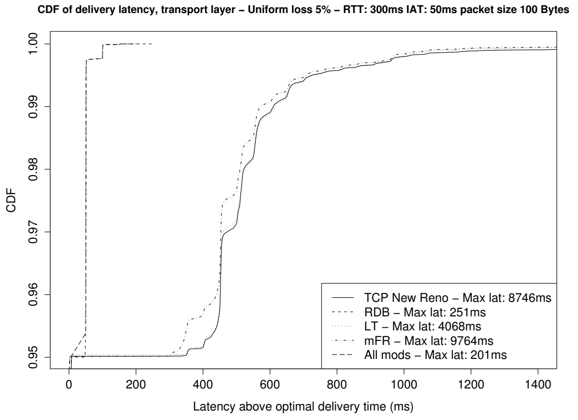

CDF of delivery latency, application layer – Uniform loss 5% – RTT: 300ms IAT: 250ms packet size 100 Bytes
CDF
0.90
0.92
0.94
0.96
0.98
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 12176ms
RDB – Max lat: 1351ms
LT – Max lat: 5778ms
mFR – Max lat: 3663ms
All mods – Max lat: 825ms

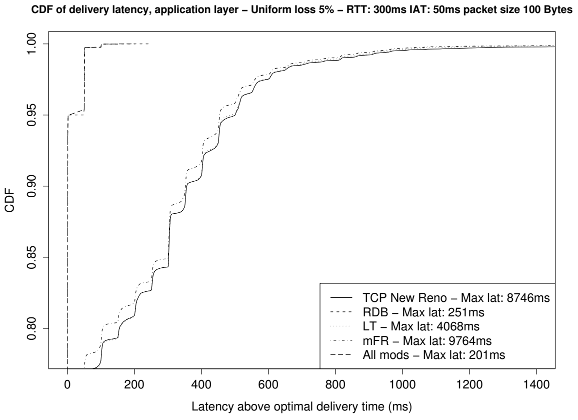

CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 300ms IAT: 25ms packet size 100 Bytes
CDF
0.95
0.96
0.97
0.98
0.99
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 4168ms
RDB – Max lat: 1016ms
LT – Max lat: 8293ms
mFR – Max lat: 4562ms
All mods – Max lat: 1025ms


CDF of delivery latency, application layer – Uniform loss 5% – RTT: 300ms IAT: 25ms packet size 100 Bytes
CDF
0.80
0.85
0.90
0.95
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 4168ms
RDB – Max lat: 1016ms
LT – Max lat: 8293ms
mFR – Max lat: 4562ms
All mods – Max lat: 1025ms

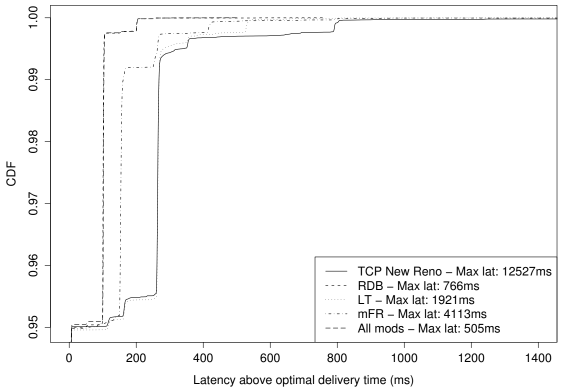

CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 300ms IAT: 500ms packet size 100 Bytes
CDF
0.96
0.97
0.98
0.99
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 6991ms
RDB – Max lat: 1550ms
LT – Max lat: 1550ms
mFR – Max lat: 3612ms
All mods – Max lat: 1048ms


CDF of delivery latency, application layer – Uniform loss 5% – RTT: 300ms IAT: 500ms packet size 100 Bytes
CDF
0.95
0.96
0.97
0.98
0.99
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 6991ms
RDB – Max lat: 1550ms
LT – Max lat: 1550ms
mFR – Max lat: 3612ms
All mods – Max lat: 1048ms

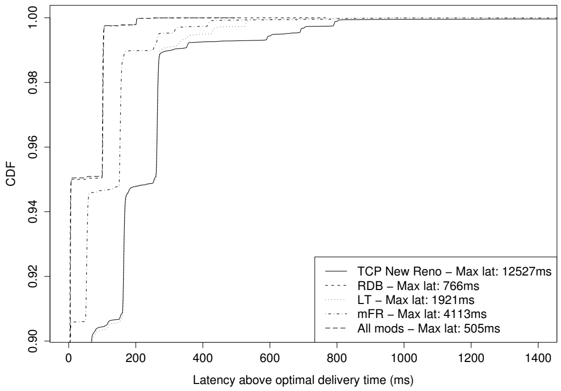

CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 300ms IAT: 50ms packet size 100 Bytes
CDF
0.95
0.96
0.97
0.98
0.99
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 8746ms
RDB – Max lat: 251ms
LT – Max lat: 4068ms
mFR – Max lat: 9764ms
All mods – Max lat: 201ms


CDF of delivery latency, application layer – Uniform loss 5% – RTT: 300ms IAT: 50ms packet size 100 Bytes
CDF
0.80
0.85
0.90
0.95
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 8746ms
RDB – Max lat: 251ms
LT – Max lat: 4068ms
mFR – Max lat: 9764ms
All mods – Max lat: 201ms

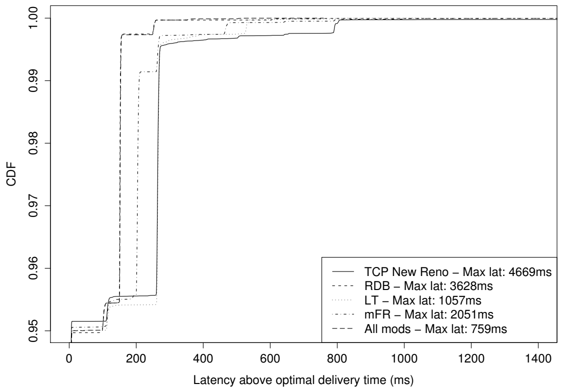
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 50ms IAT: 100ms packet size 100 Bytes



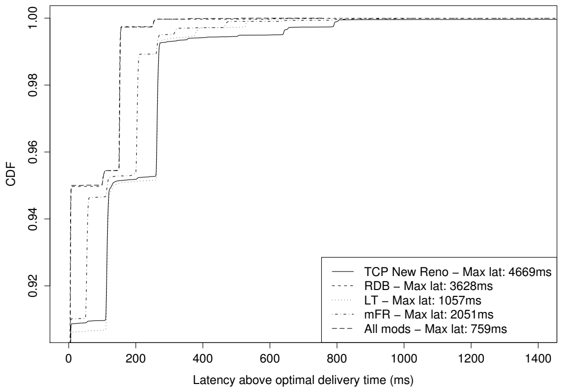
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 50ms IAT: 100ms packet size 100 Bytes



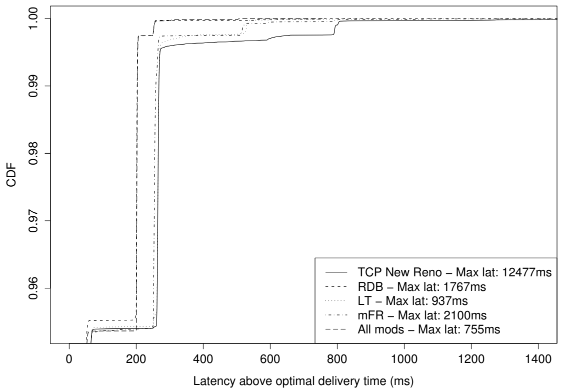
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 50ms IAT: 150ms packet size 100 Bytes



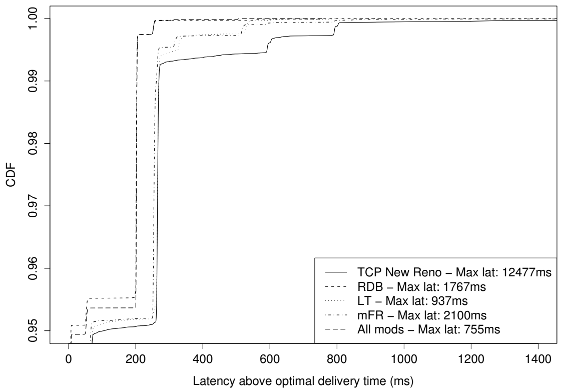
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 50ms IAT: 150ms packet size 100 Bytes



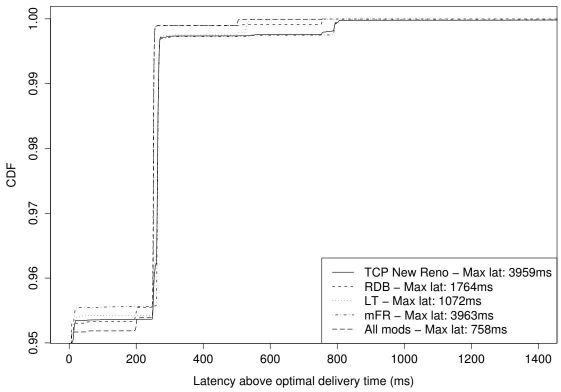
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 50ms IAT: 200ms packet size 100 Bytes



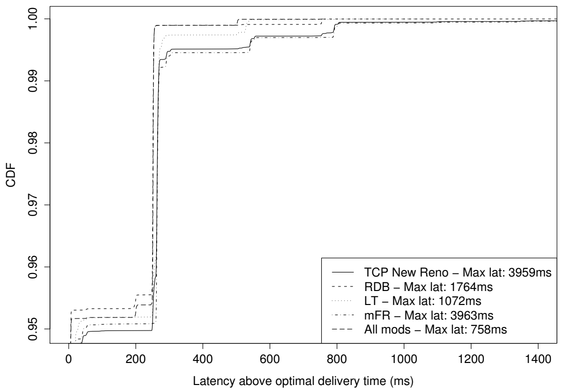
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 50ms IAT: 200ms packet size 100 Bytes



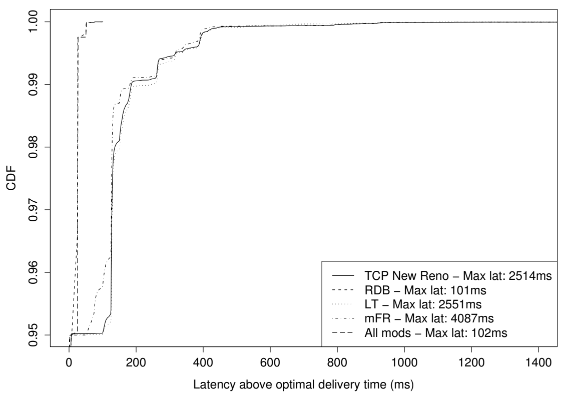
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 50ms IAT: 250ms packet size 100 Bytes



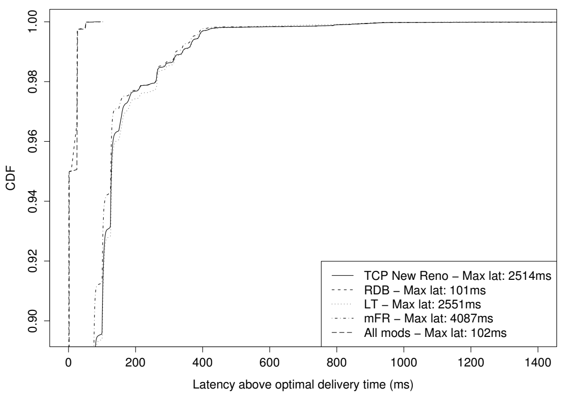
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 50ms IAT: 250ms packet size 100 Bytes



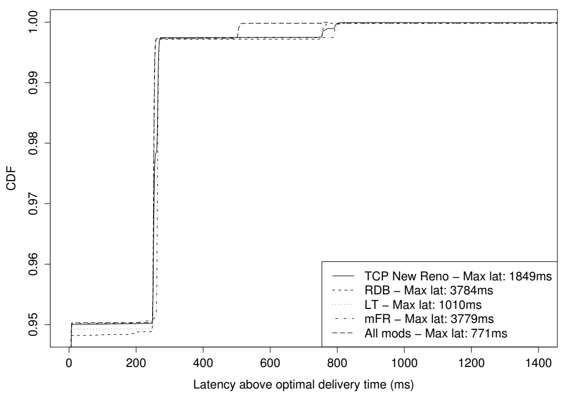
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 50ms IAT: 25ms packet size 100 Bytes



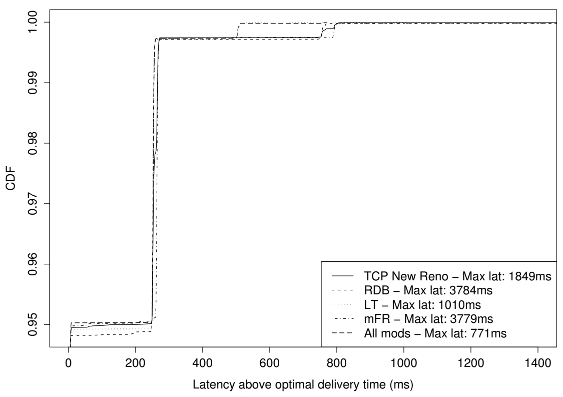
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 50ms IAT: 25ms packet size 100 Bytes



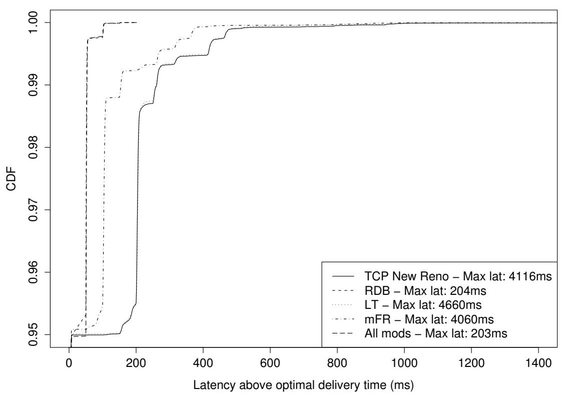
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 50ms IAT: 500ms packet size 100 Bytes



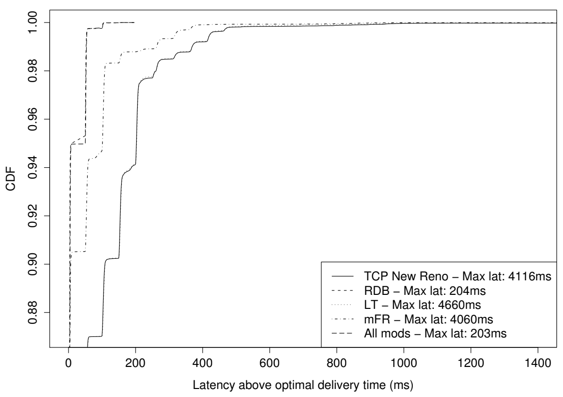
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 50ms IAT: 500ms packet size 100 Bytes



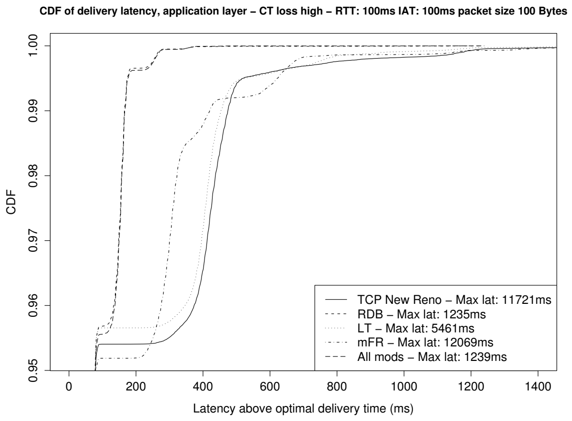
CDF of delivery latency, transport layer – Uniform loss 5% – RTT: 50ms IAT: 50ms packet size 100 Bytes

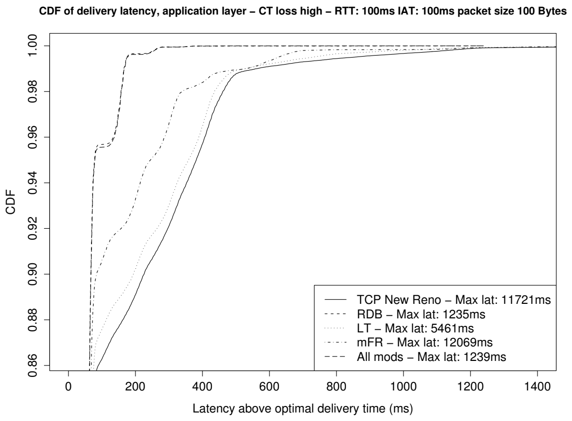


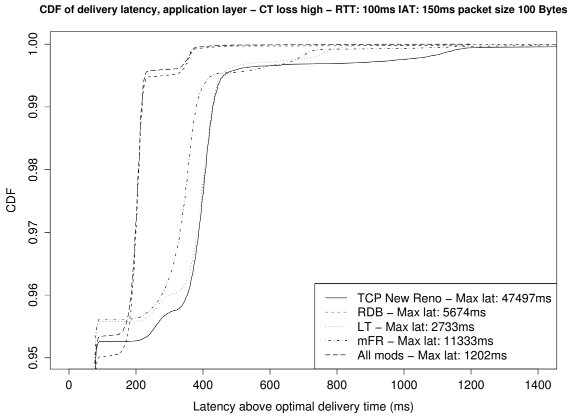
CDF of delivery latency, application layer – Uniform loss 5% – RTT: 50ms IAT: 50ms packet size 100 Bytes

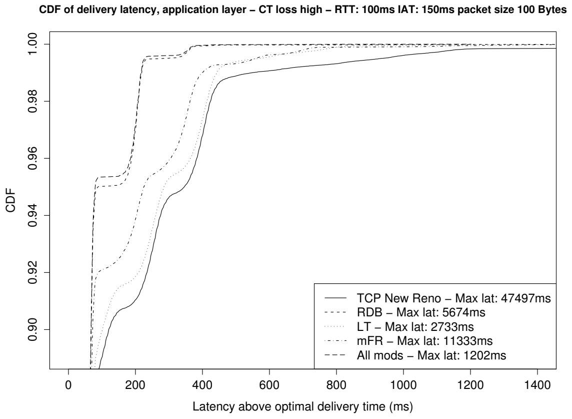


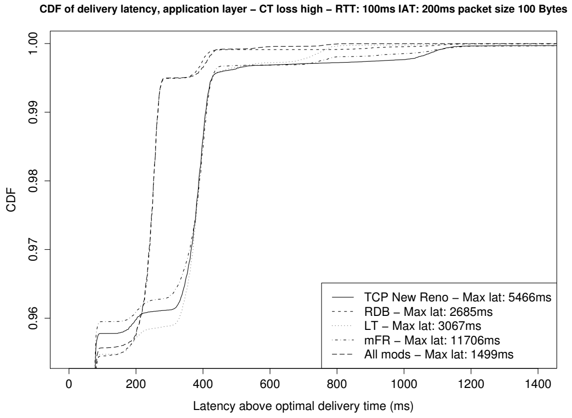
D.2 Cross-traffic loss - high loss rate

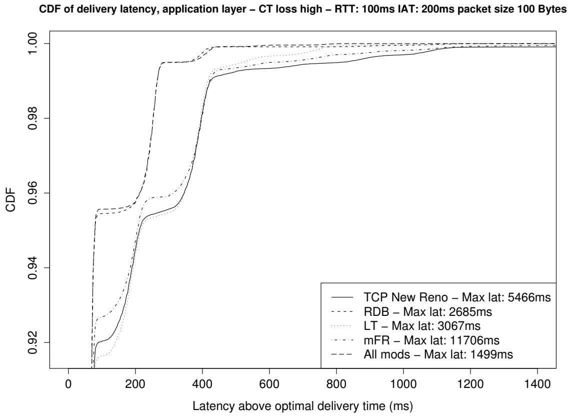

CDF of delivery latency, application layer – CT loss high – RTT: 100ms IAT: 100ms packet size 100 Bytes
CDF
0.95
0.96
0.97
0.98
0.99
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 11721ms
RDB – Max lat: 1235ms
LT – Max lat: 5461ms
mFR – Max lat: 12069ms
All mods – Max lat: 1239ms

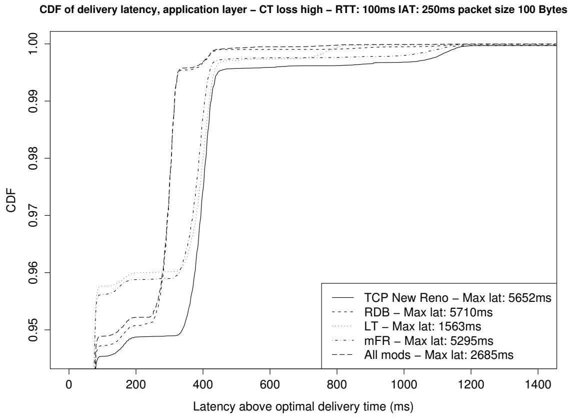

CDF of delivery latency, application layer – CT loss high – RTT: 100ms IAT: 100ms packet size 100 Bytes
CDF
0.86
0.88
0.90
0.92
0.94
0.96
0.98
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 11721ms
RDB – Max lat: 1235ms
LT – Max lat: 5461ms
mFR – Max lat: 12069ms
All mods – Max lat: 1239ms

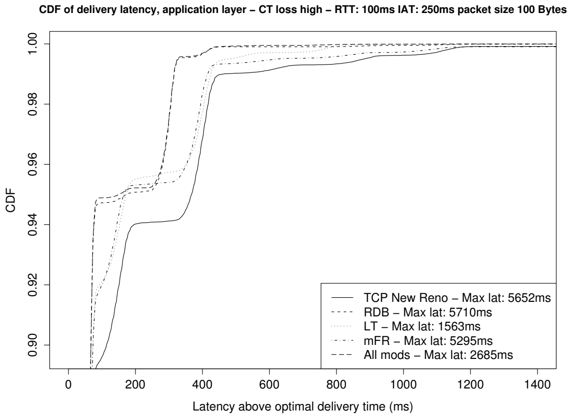

CDF of delivery latency, application layer – CT loss high – RTT: 100ms IAT: 150ms packet size 100 Bytes
CDF
0.95
0.96
0.97
0.98
0.99
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 47497ms
RDB – Max lat: 5674ms
LT – Max lat: 2733ms
mFR – Max lat: 11333ms
All mods – Max lat: 1202ms


CDF of delivery latency, application layer – CT loss high – RTT: 100ms IAT: 150ms packet size 100 Bytes
CDF
0.90
0.92
0.94
0.96
0.98
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 47497ms
RDB – Max lat: 5674ms
LT – Max lat: 2733ms
mFR – Max lat: 11333ms
All mods – Max lat: 1202ms

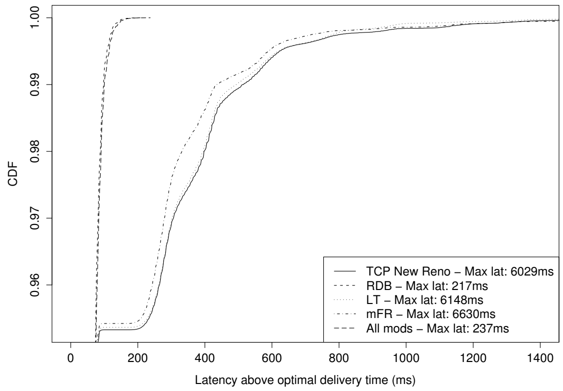

CDF of delivery latency, application layer – CT loss high – RTT: 100ms IAT: 200ms packet size 100 Bytes
CDF
0.96
0.97
0.98
0.99
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 5466ms
RDB – Max lat: 2685ms
LT – Max lat: 3067ms
mFR – Max lat: 11706ms
All mods – Max lat: 1499ms


CDF of delivery latency, application layer – CT loss high – RTT: 100ms IAT: 200ms packet size 100 Bytes
CDF
0.92
0.94
0.96
0.98
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 5466ms
RDB – Max lat: 2685ms
LT – Max lat: 3067ms
mFR – Max lat: 11706ms
All mods – Max lat: 1499ms

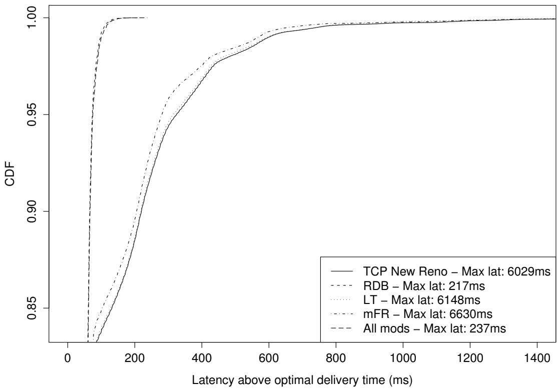

CDF of delivery latency, application layer – CT loss high – RTT: 100ms IAT: 250ms packet size 100 Bytes
CDF
0.95
0.96
0.97
0.98
0.99
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 5652ms
RDB – Max lat: 5710ms
LT – Max lat: 1563ms
mFR – Max lat: 5295ms
All mods – Max lat: 2685ms


CDF of delivery latency, application layer – CT loss high – RTT: 100ms IAT: 250ms packet size 100 Bytes
CDF
0.90
0.92
0.94
0.96
0.98
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 5652ms
RDB – Max lat: 5710ms
LT – Max lat: 1563ms
mFR – Max lat: 5295ms
All mods – Max lat: 2685ms

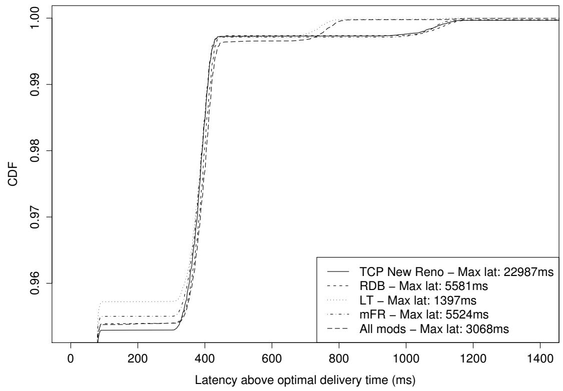
CDF of delivery latency, application layer – CT loss high – RTT: 100ms IAT: 25ms packet size 100 Bytes



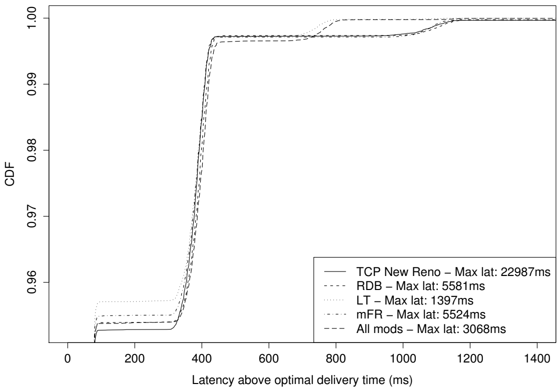
CDF of delivery latency, application layer – CT loss high – RTT: 100ms IAT: 25ms packet size 100 Bytes



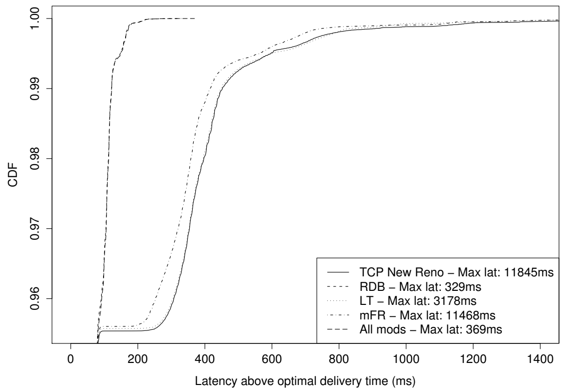
CDF of delivery latency, application layer – CT loss high – RTT: 100ms IAT: 500ms packet size 100 Bytes



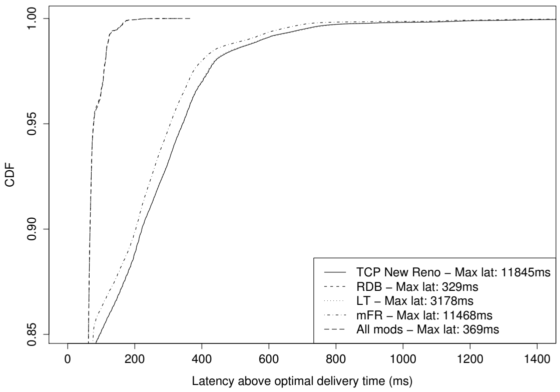
CDF of delivery latency, application layer – CT loss high – RTT: 100ms IAT: 500ms packet size 100 Bytes



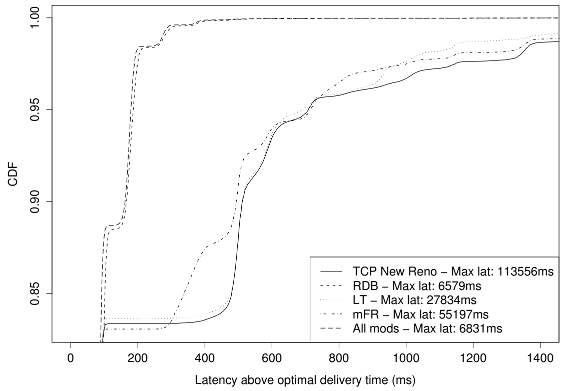
CDF of delivery latency, application layer – CT loss high – RTT: 100ms IAT: 50ms packet size 100 Bytes



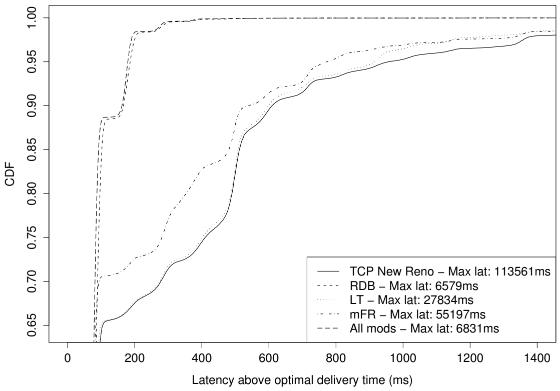
CDF of delivery latency, application layer – CT loss high – RTT: 100ms IAT: 50ms packet size 100 Bytes

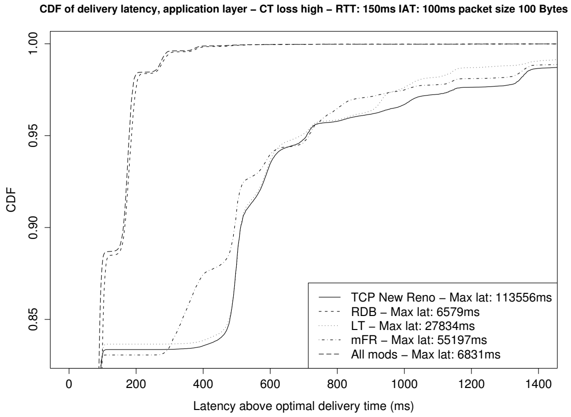


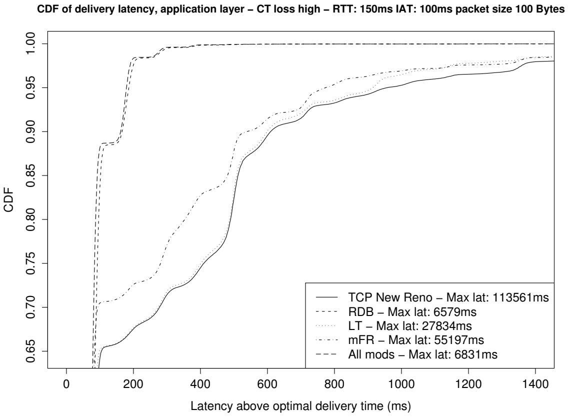
CDF of delivery latency, application layer – CT loss high – RTT: 150ms IAT: 100ms packet size 100 Bytes

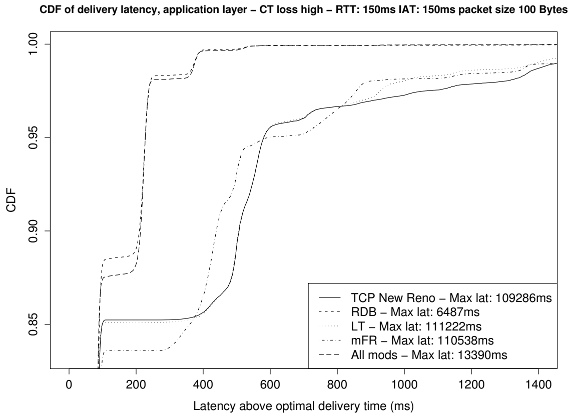


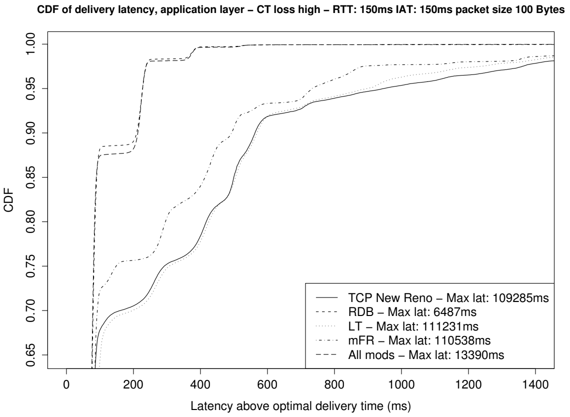
CDF of delivery latency, application layer – CT loss high – RTT: 150ms IAT: 100ms packet size 100 Bytes

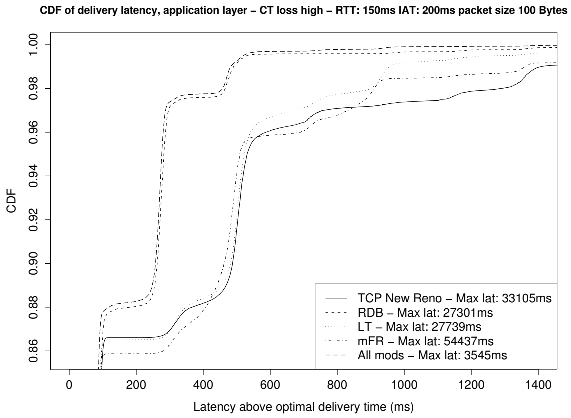


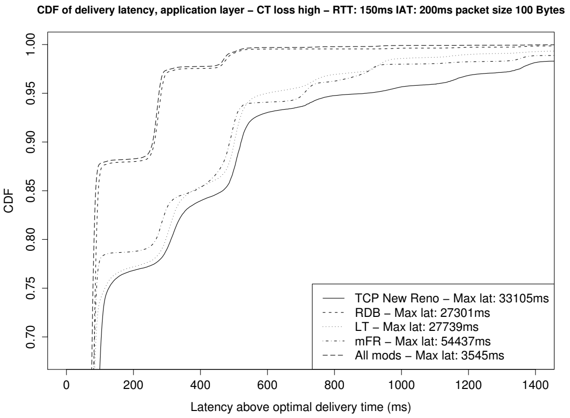

CDF of delivery latency, application layer – CT loss high – RTT: 150ms IAT: 100ms packet size 100 Bytes

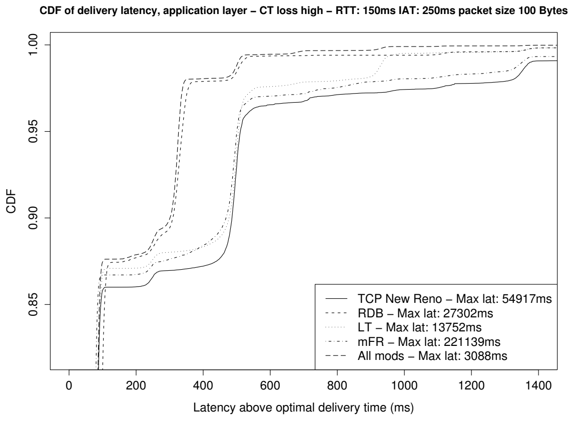

CDF of delivery latency, application layer – CT loss high – RTT: 150ms IAT: 100ms packet size 100 Bytes

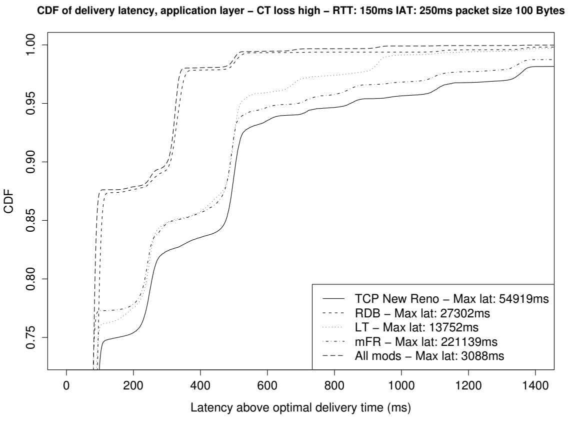

CDF of delivery latency, application layer – CT loss high – RTT: 150ms IAT: 150ms packet size 100 Bytes

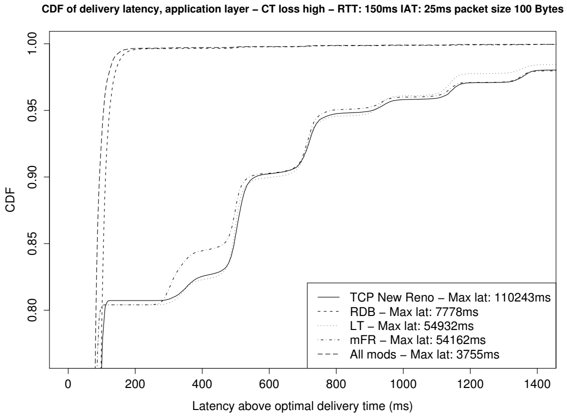

CDF of delivery latency, application layer – CT loss high – RTT: 150ms IAT: 150ms packet size 100 Bytes

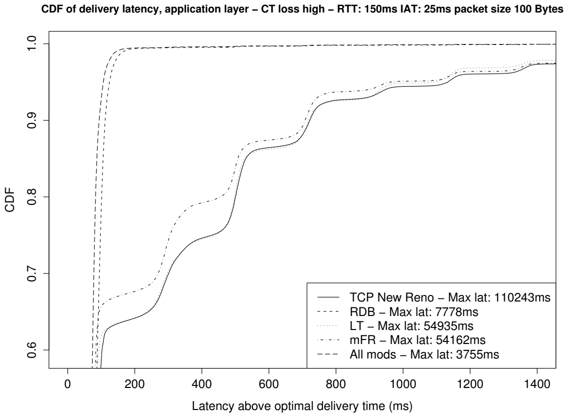

CDF of delivery latency, application layer – CT loss high – RTT: 150ms IAT: 200ms packet size 100 Bytes

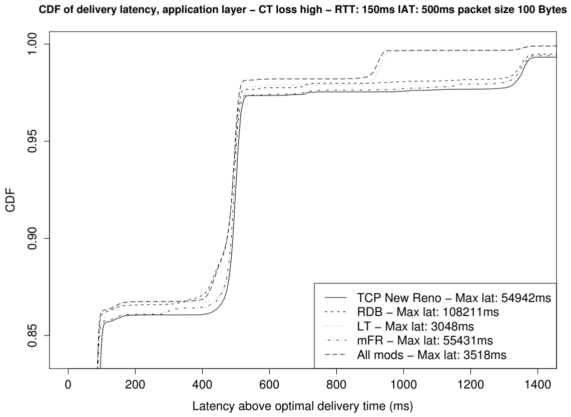

CDF of delivery latency, application layer – CT loss high – RTT: 150ms IAT: 200ms packet size 100 Bytes

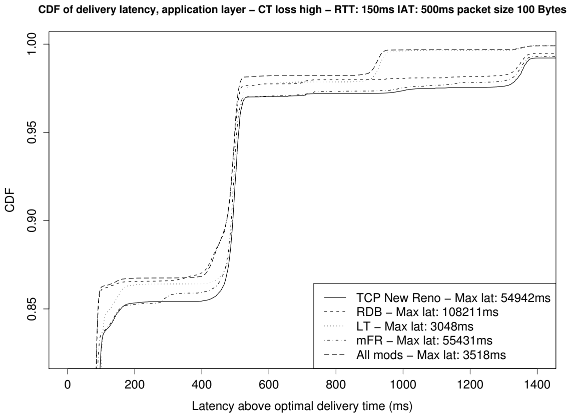

CDF of delivery latency, application layer – CT loss high – RTT: 150ms IAT: 250ms packet size 100 Bytes

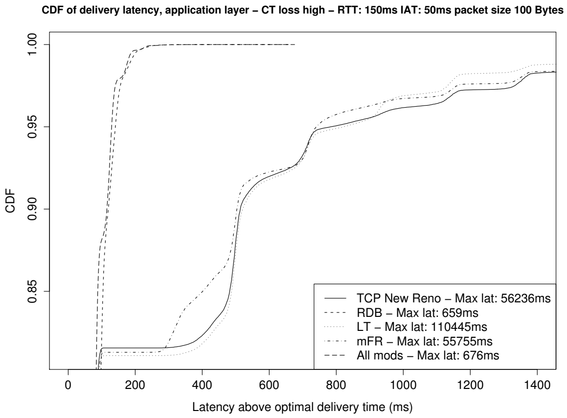

CDF of delivery latency, application layer – CT loss high – RTT: 150ms IAT: 250ms packet size 100 Bytes

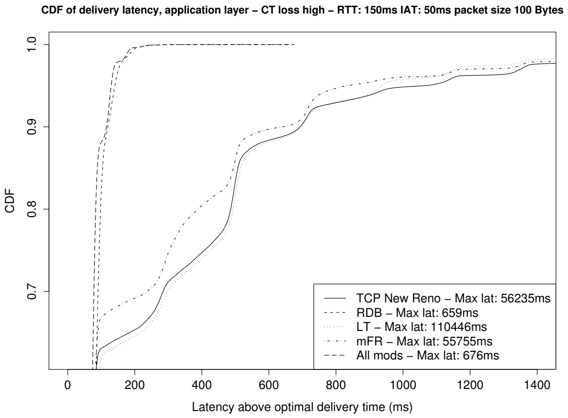

CDF of delivery latency, application layer – CT loss high – RTT: 150ms IAT: 25ms packet size 100 Bytes
CDF
0.80
0.85
0.90
0.95
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 110243ms
RDB – Max lat: 7778ms
LT – Max lat: 54932ms
mFR – Max lat: 54162ms
All mods – Max lat: 3755ms

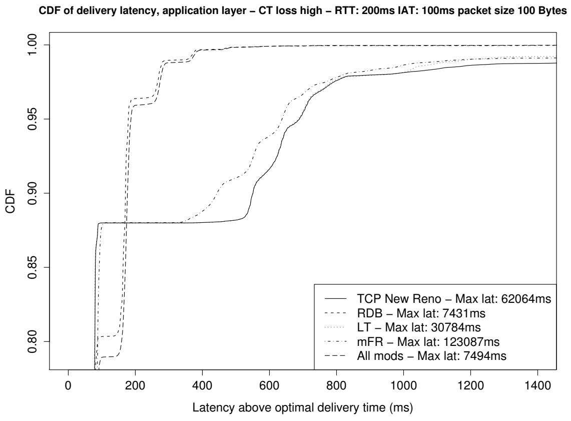

CDF of delivery latency, application layer – CT loss high – RTT: 150ms IAT: 25ms packet size 100 Bytes
CDF
0.6
0.7
0.8
0.9
1.0
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 110243ms
RDB – Max lat: 7778ms
LT – Max lat: 54935ms
mFR – Max lat: 54162ms
All mods – Max lat: 3755ms

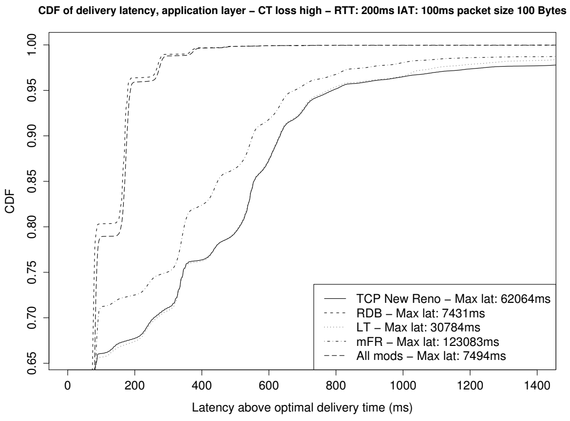

CDF of delivery latency, application layer – CT loss high – RTT: 150ms IAT: 500ms packet size 100 Bytes
CDF
0.85
0.90
0.95
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 54942ms
RDB – Max lat: 108211ms
LT – Max lat: 3048ms
mFR – Max lat: 55431ms
All mods – Max lat: 3518ms

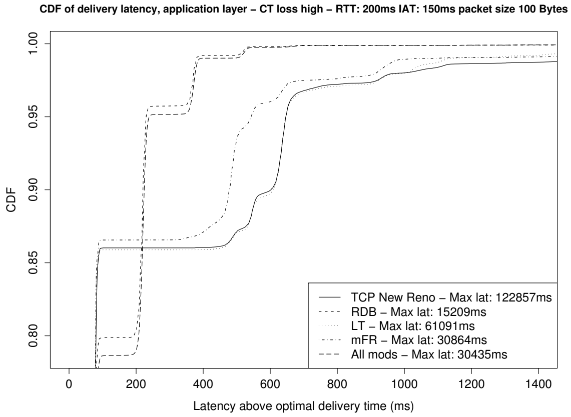

CDF of delivery latency, application layer – CT loss high – RTT: 150ms IAT: 500ms packet size 100 Bytes
CDF
0.85
0.90
0.95
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 54942ms
RDB – Max lat: 108211ms
LT – Max lat: 3048ms
mFR – Max lat: 55431ms
All mods – Max lat: 3518ms

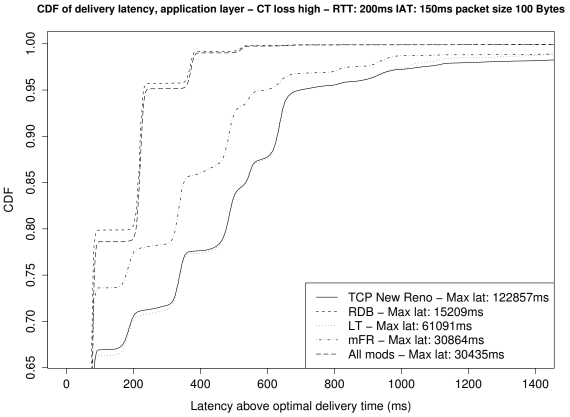

CDF of delivery latency, application layer – CT loss high – RTT: 150ms IAT: 50ms packet size 100 Bytes
CDF
0.85
0.90
0.95
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 56236ms
RDB – Max lat: 659ms
LT – Max lat: 110445ms
mFR – Max lat: 55755ms
All mods – Max lat: 676ms

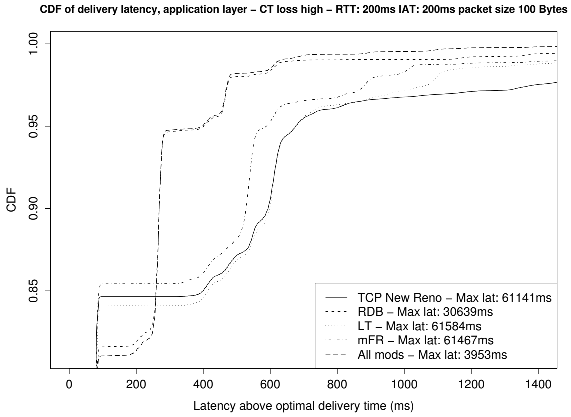

CDF of delivery latency, application layer – CT loss high – RTT: 150ms IAT: 50ms packet size 100 Bytes
CDF
0.7
0.8
0.9
1.0
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 56235ms
RDB – Max lat: 659ms
LT – Max lat: 110446ms
mFR – Max lat: 55755ms
All mods – Max lat: 676ms

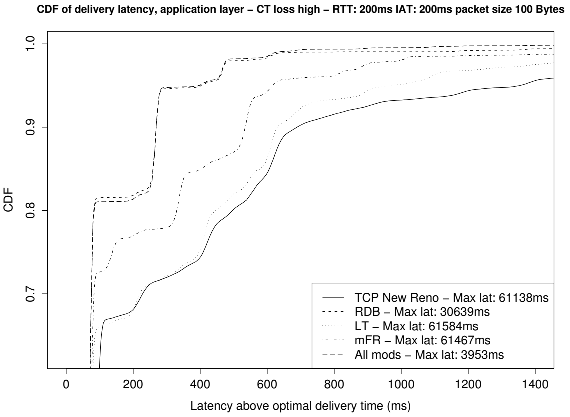

CDF of delivery latency, application layer – CT loss high – RTT: 200ms IAT: 100ms packet size 100 Bytes
CDF
0.80
0.85
0.90
0.95
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 62064ms
RDB – Max lat: 7431ms
LT – Max lat: 30784ms
mFR – Max lat: 123087ms
All mods – Max lat: 7494ms

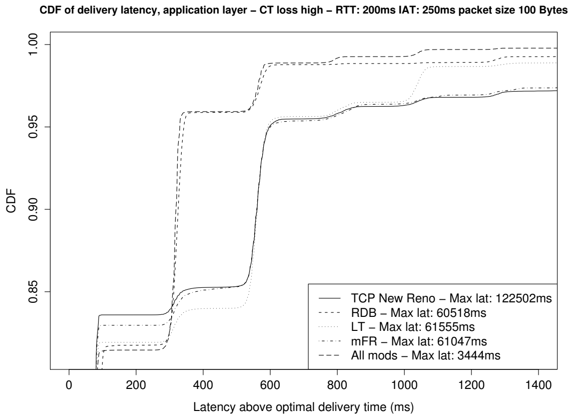

CDF of delivery latency, application layer – CT loss high – RTT: 200ms IAT: 100ms packet size 100 Bytes
CDF
0.65
0.70
0.75
0.80
0.85
0.90
0.95
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 62064ms
RDB – Max lat: 7431ms
LT – Max lat: 30784ms
mFR – Max lat: 123083ms
All mods – Max lat: 7494ms

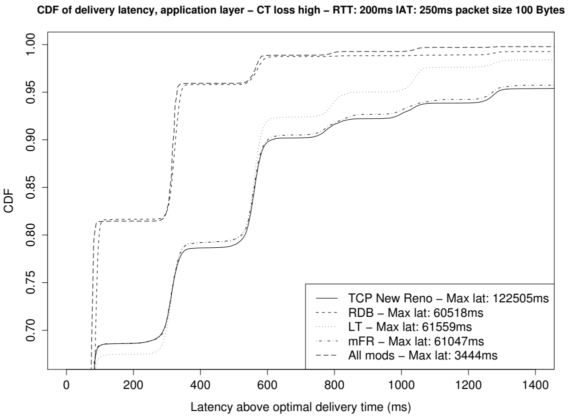

CDF of delivery latency, application layer – CT loss high – RTT: 200ms IAT: 150ms packet size 100 Bytes
CDF
0.80
0.85
0.90
0.95
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 122857ms
RDB – Max lat: 15209ms
LT – Max lat: 61091ms
mFR – Max lat: 30864ms
All mods – Max lat: 30435ms

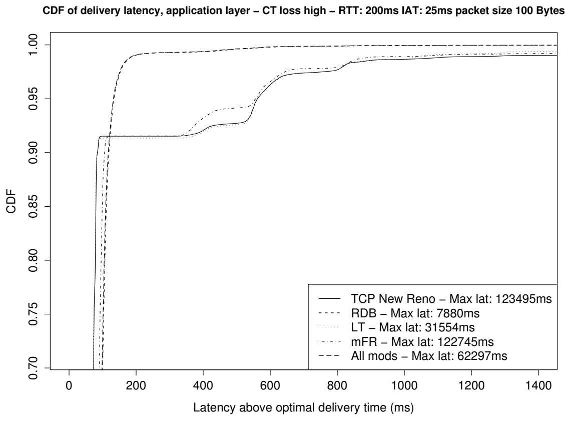

CDF of delivery latency, application layer – CT loss high – RTT: 200ms IAT: 150ms packet size 100 Bytes
CDF
0.65
0.70
0.75
0.80
0.85
0.90
0.95
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 122857ms
RDB – Max lat: 15209ms
LT – Max lat: 61091ms
mFR – Max lat: 30864ms
All mods – Max lat: 30435ms

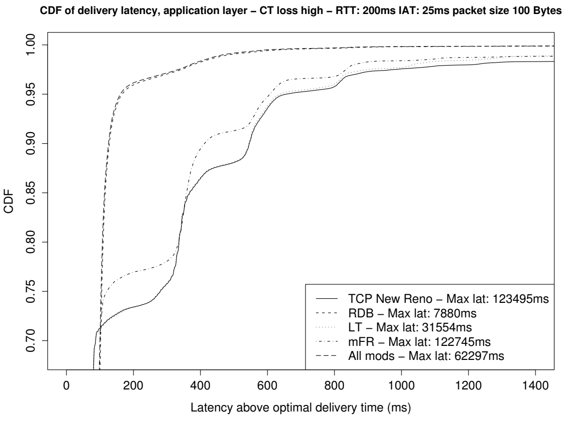

CDF of delivery latency, application layer – CT loss high – RTT: 200ms IAT: 200ms packet size 100 Bytes
CDF
0.85
0.90
0.95
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 61141ms
RDB – Max lat: 30639ms
LT – Max lat: 61584ms
mFR – Max lat: 61467ms
All mods – Max lat: 3953ms


CDF of delivery latency, application layer – CT loss high – RTT: 200ms IAT: 200ms packet size 100 Bytes
CDF
0.7
0.8
0.9
1.0
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 61138ms
RDB – Max lat: 30639ms
LT – Max lat: 61584ms
mFR – Max lat: 61467ms
All mods – Max lat: 3953ms

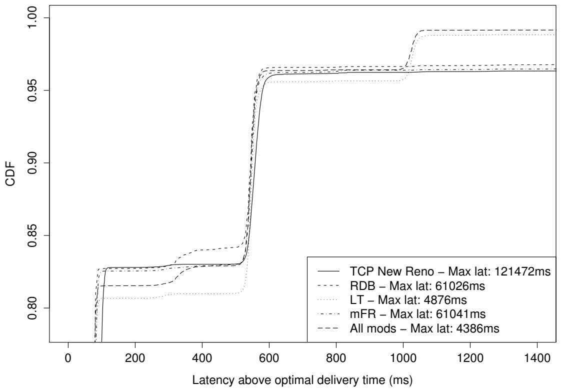

CDF of delivery latency, application layer – CT loss high – RTT: 200ms IAT: 250ms packet size 100 Bytes
CDF
0.85
0.90
0.95
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 122502ms
RDB – Max lat: 60518ms
LT – Max lat: 61555ms
mFR – Max lat: 61047ms
All mods – Max lat: 3444ms


CDF of delivery latency, application layer – CT loss high – RTT: 200ms IAT: 250ms packet size 100 Bytes
CDF
0.70
0.75
0.80
0.85
0.90
0.95
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 122505ms
RDB – Max lat: 60518ms
LT – Max lat: 61559ms
mFR – Max lat: 61047ms
All mods – Max lat: 3444ms

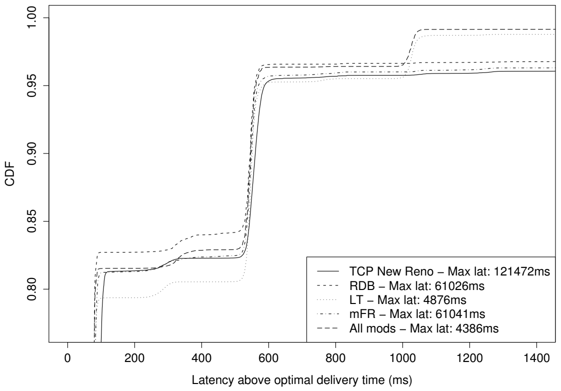

CDF of delivery latency, application layer – CT loss high – RTT: 200ms IAT: 25ms packet size 100 Bytes
CDF
0.70
0.75
0.80
0.85
0.90
0.95
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 123495ms
RDB – Max lat: 7880ms
LT – Max lat: 31554ms
mFR – Max lat: 122745ms
All mods – Max lat: 62297ms


CDF of delivery latency, application layer – CT loss high – RTT: 200ms IAT: 25ms packet size 100 Bytes
CDF
0.70
0.75
0.80
0.85
0.90
0.95
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 123495ms
RDB – Max lat: 7880ms
LT – Max lat: 31554ms
mFR – Max lat: 122745ms
All mods – Max lat: 62297ms

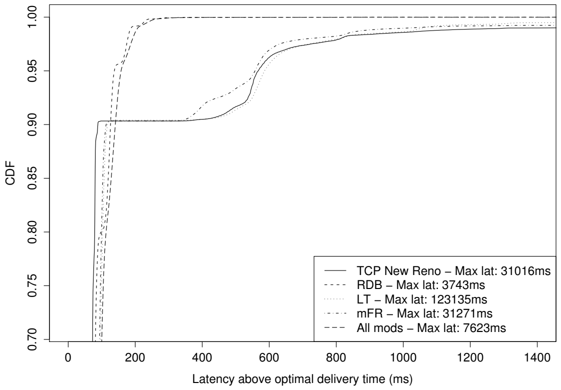
CDF of delivery latency, application layer – CT loss high – RTT: 200ms IAT: 500ms packet size 100 Bytes



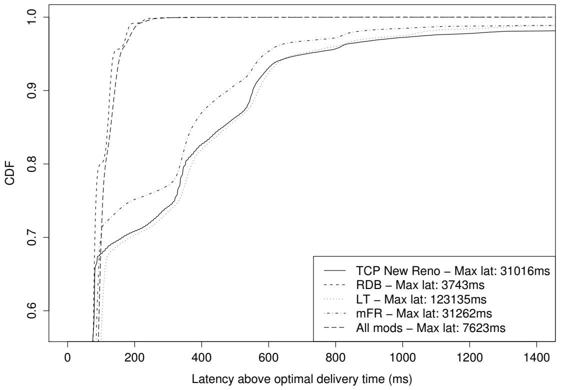
CDF of delivery latency, application layer – CT loss high – RTT: 200ms IAT: 500ms packet size 100 Bytes



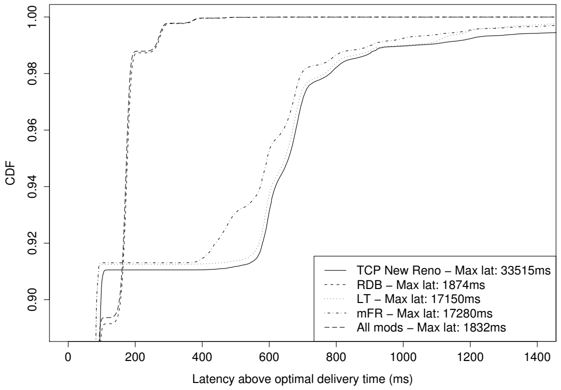
CDF of delivery latency, application layer – CT loss high – RTT: 200ms IAT: 50ms packet size 100 Bytes



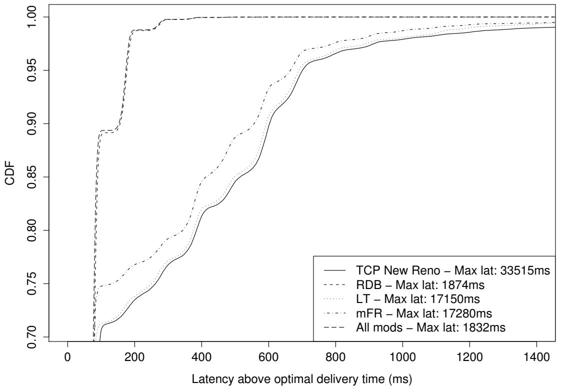
CDF of delivery latency, application layer – CT loss high – RTT: 200ms IAT: 50ms packet size 100 Bytes



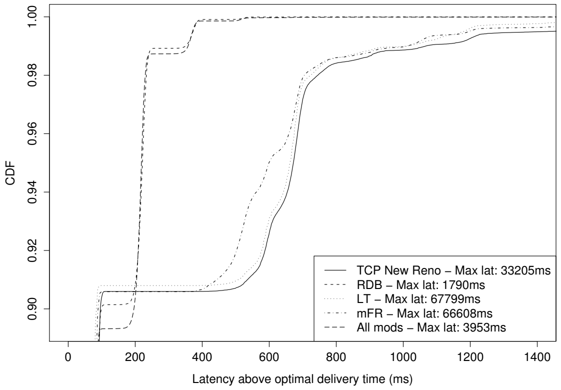
CDF of delivery latency, application layer – CT loss high – RTT: 250ms IAT: 100ms packet size 100 Bytes



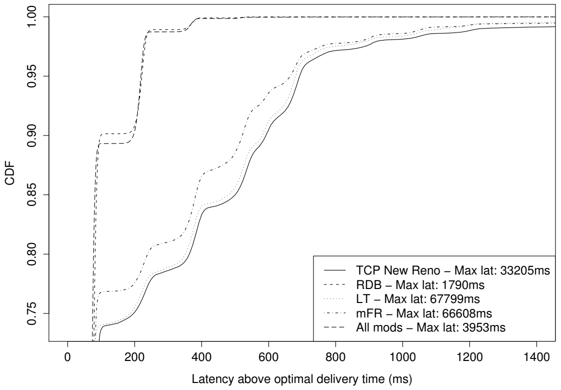
CDF of delivery latency, application layer – CT loss high – RTT: 250ms IAT: 100ms packet size 100 Bytes

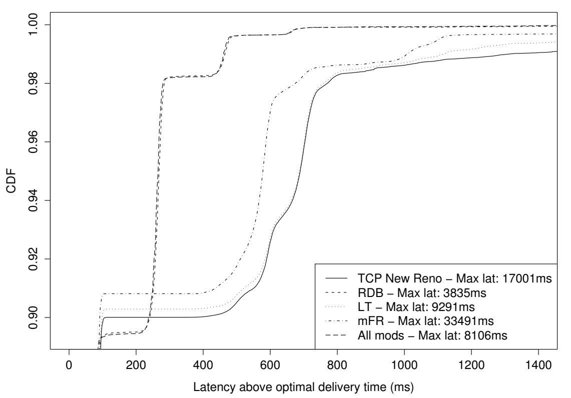


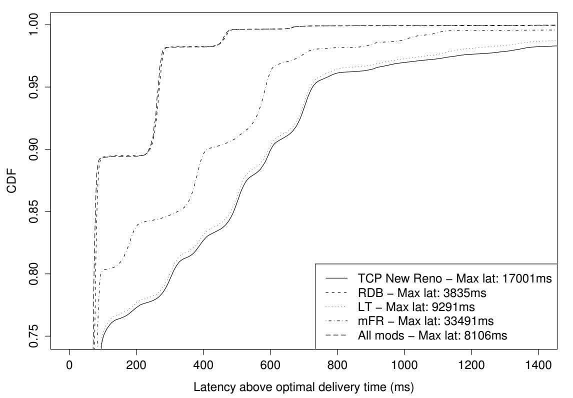
CDF of delivery latency, application layer – CT loss high – RTT: 250ms IAT: 150ms packet size 100 Bytes

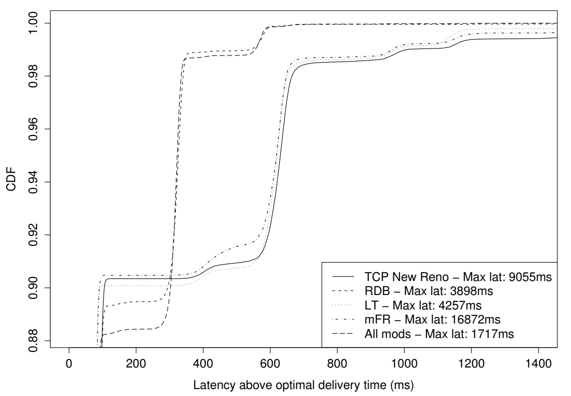


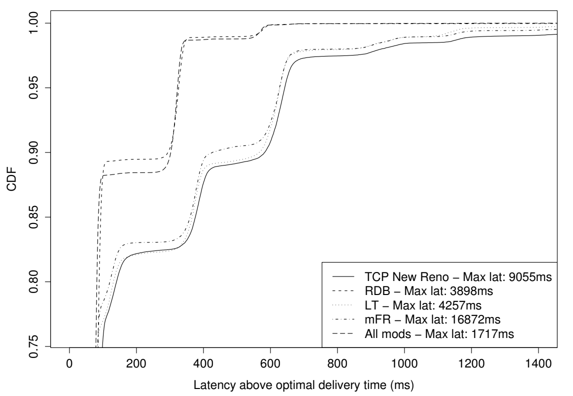
CDF of delivery latency, application layer – CT loss high – RTT: 250ms IAT: 150ms packet size 100 Bytes

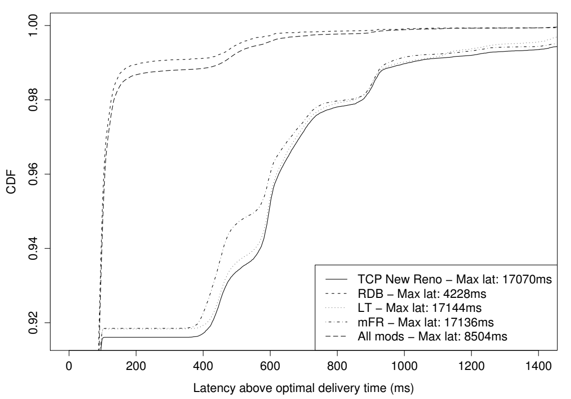


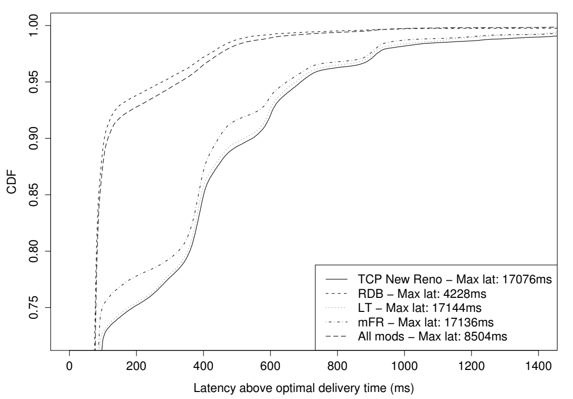

CDF of delivery latency, application layer – CT loss high – RTT: 250ms IAT: 200ms packet size 100 Bytes

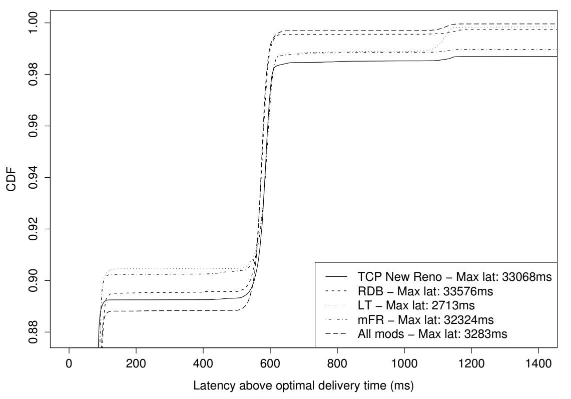

CDF of delivery latency, application layer – CT loss high – RTT: 250ms IAT: 200ms packet size 100 Bytes

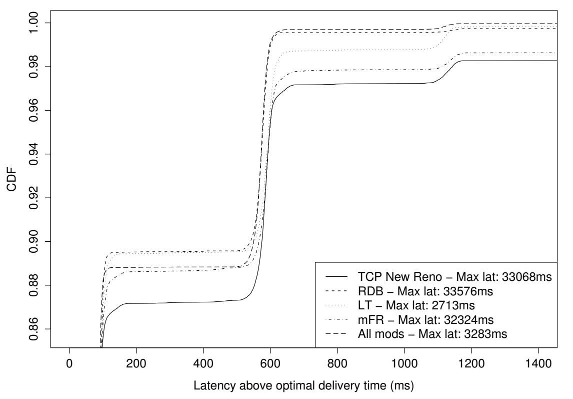

CDF of delivery latency, application layer – CT loss high – RTT: 250ms IAT: 250ms packet size 100 Bytes

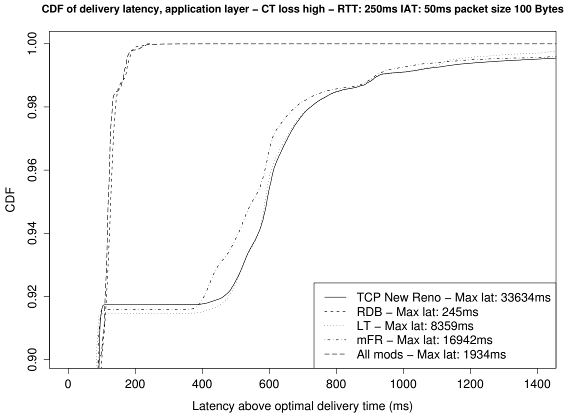

CDF of delivery latency, application layer – CT loss high – RTT: 250ms IAT: 250ms packet size 100 Bytes

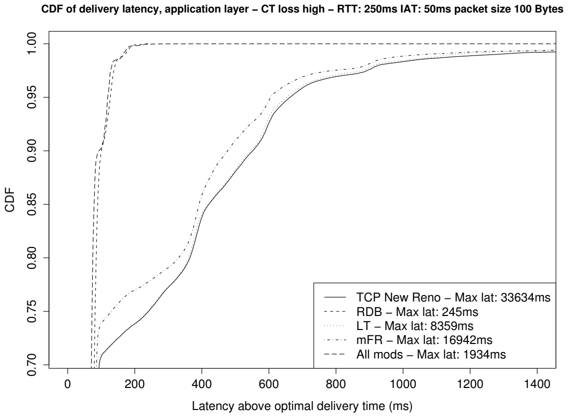

CDF of delivery latency, application layer – CT loss high – RTT: 250ms IAT: 25ms packet size 100 Bytes

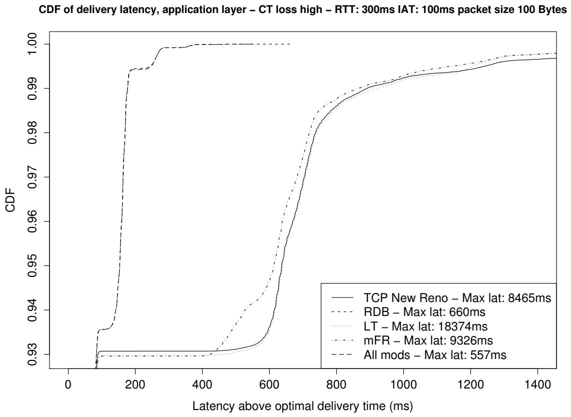

CDF of delivery latency, application layer – CT loss high – RTT: 250ms IAT: 25ms packet size 100 Bytes

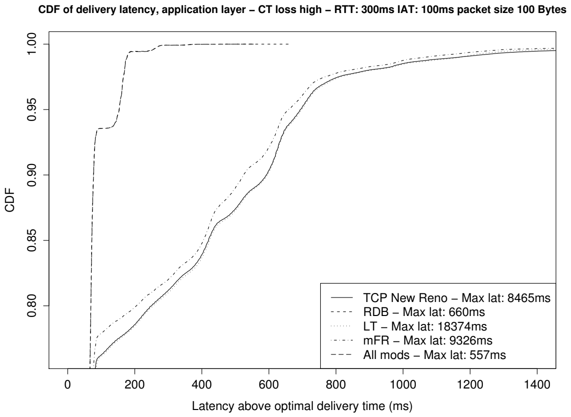

CDF of delivery latency, application layer – CT loss high – RTT: 250ms IAT: 500ms packet size 100 Bytes

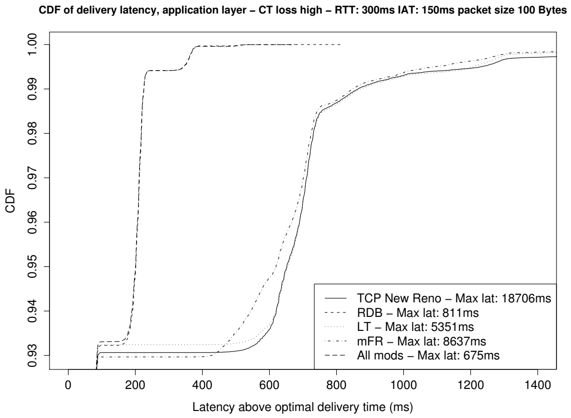

CDF of delivery latency, application layer – CT loss high – RTT: 250ms IAT: 500ms packet size 100 Bytes

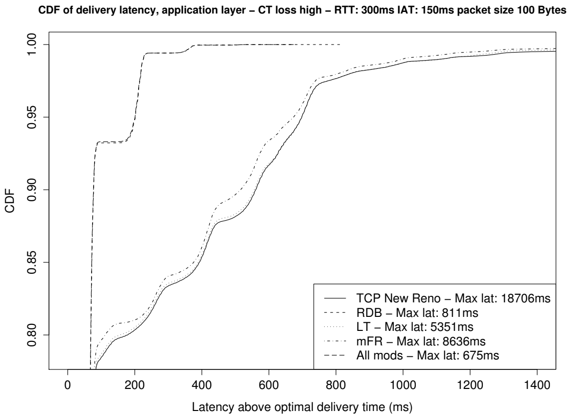

CDF of delivery latency, application layer – CT loss high – RTT: 250ms IAT: 50ms packet size 100 Bytes

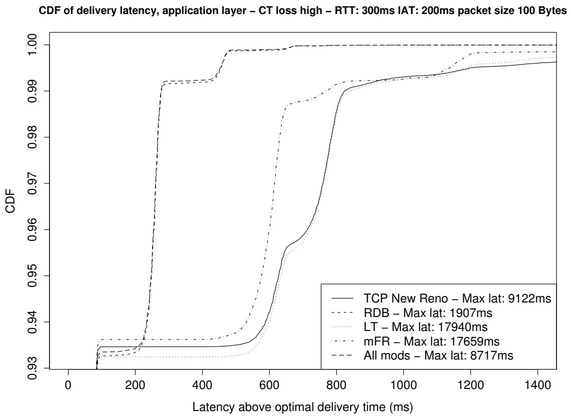

CDF of delivery latency, application layer – CT loss high – RTT: 250ms IAT: 50ms packet size 100 Bytes

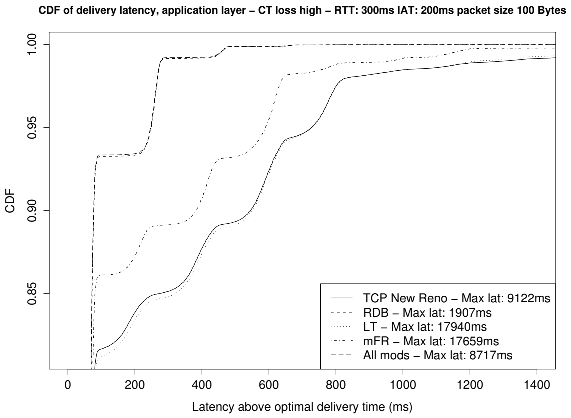

CDF of delivery latency, application layer – CT loss high – RTT: 300ms IAT: 100ms packet size 100 Bytes


CDF of delivery latency, application layer – CT loss high – RTT: 300ms IAT: 100ms packet size 100 Bytes

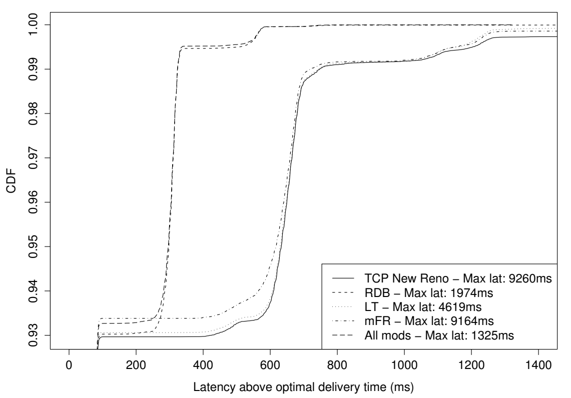

CDF of delivery latency, application layer – CT loss high – RTT: 300ms IAT: 150ms packet size 100 Bytes


CDF of delivery latency, application layer – CT loss high – RTT: 300ms IAT: 150ms packet size 100 Bytes

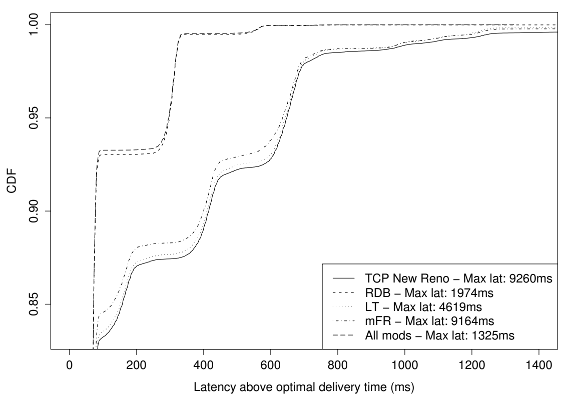

CDF of delivery latency, application layer – CT loss high – RTT: 300ms IAT: 200ms packet size 100 Bytes


CDF of delivery latency, application layer – CT loss high – RTT: 300ms IAT: 200ms packet size 100 Bytes

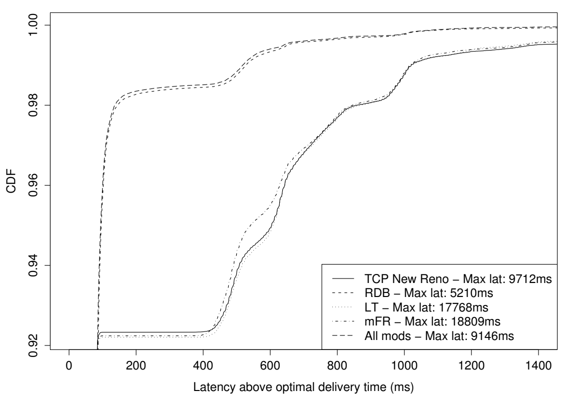
CDF of delivery latency, application layer – CT loss high – RTT: 300ms IAT: 250ms packet size 100 Bytes



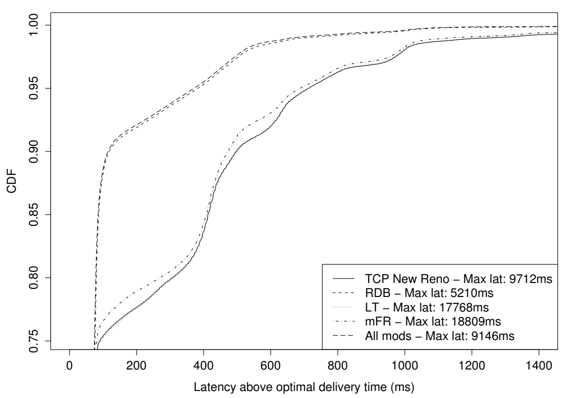
CDF of delivery latency, application layer – CT loss high – RTT: 300ms IAT: 250ms packet size 100 Bytes



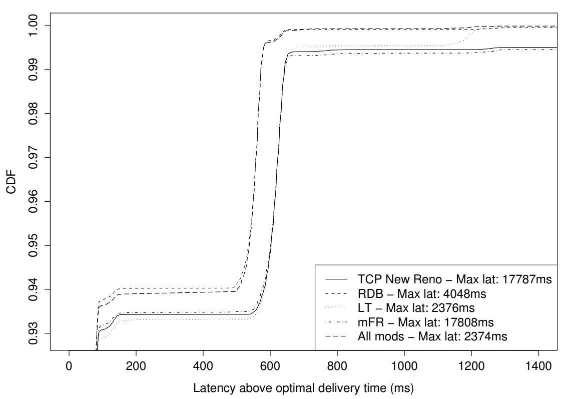
CDF of delivery latency, application layer – CT loss high – RTT: 300ms IAT: 25ms packet size 100 Bytes



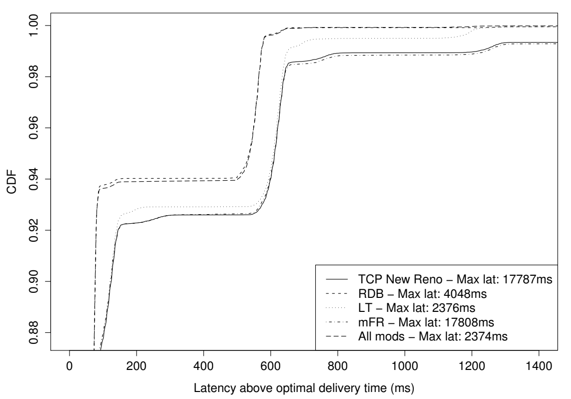
CDF of delivery latency, application layer – CT loss high – RTT: 300ms IAT: 25ms packet size 100 Bytes



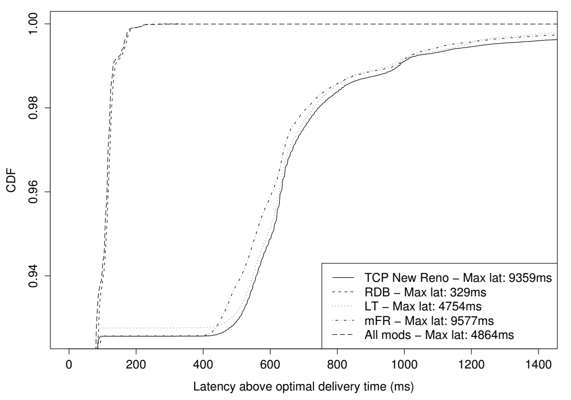
CDF of delivery latency, application layer – CT loss high – RTT: 300ms IAT: 500ms packet size 100 Bytes



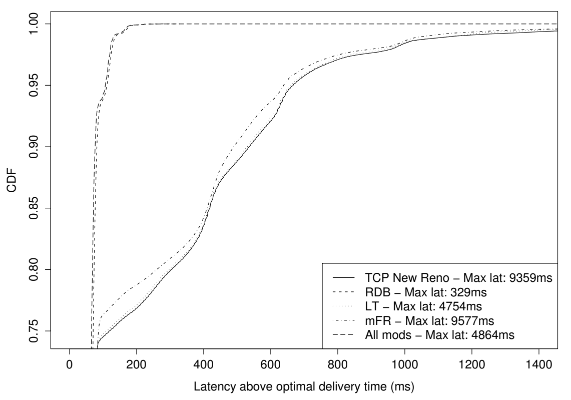
CDF of delivery latency, application layer – CT loss high – RTT: 300ms IAT: 500ms packet size 100 Bytes

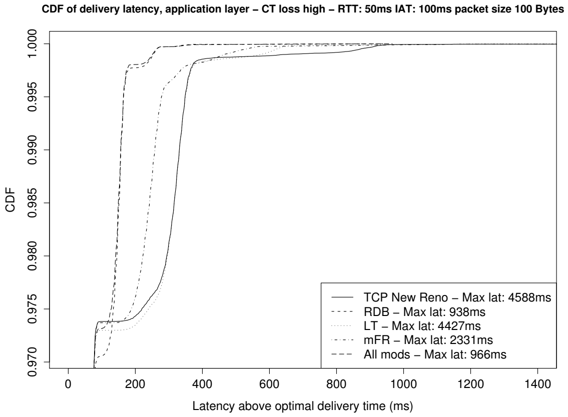


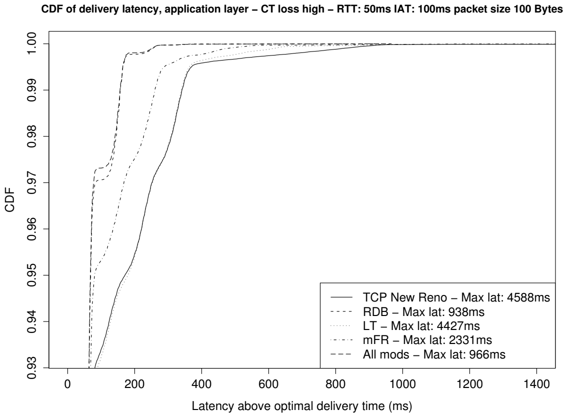
CDF of delivery latency, application layer – CT loss high – RTT: 300ms IAT: 50ms packet size 100 Bytes

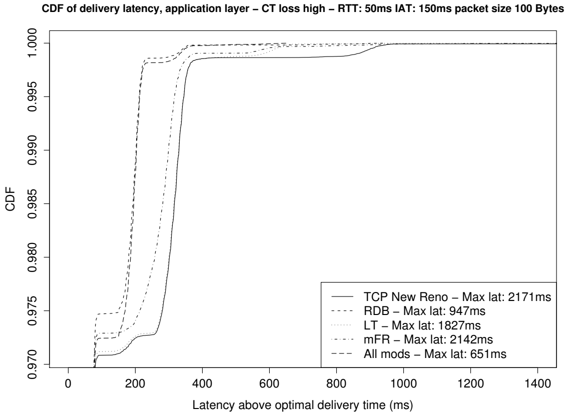


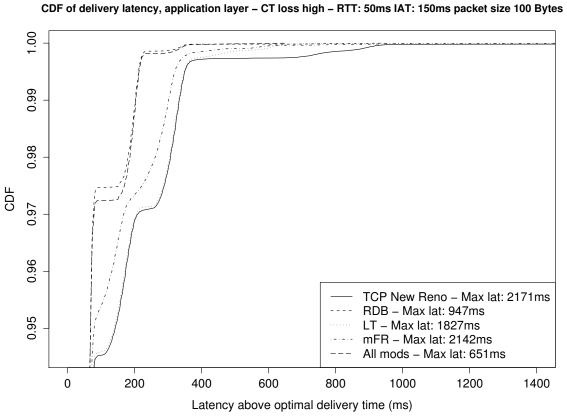
CDF of delivery latency, application layer – CT loss high – RTT: 300ms IAT: 50ms packet size 100 Bytes

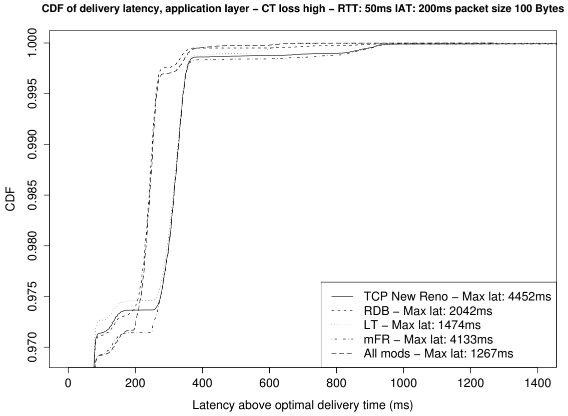


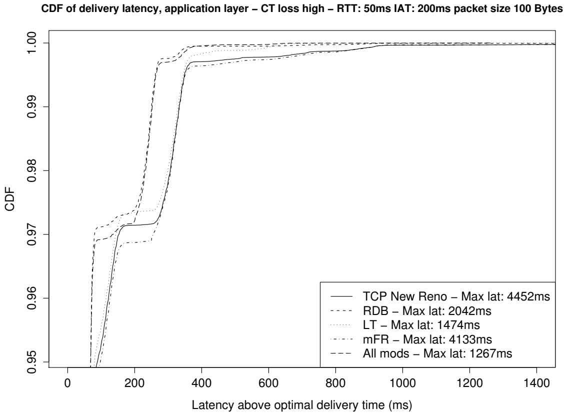

CDF of delivery latency, application layer – CT loss high – RTT: 50ms IAT: 100ms packet size 100 Bytes
CDF
0.970
0.975
0.980
0.985
0.990
0.995
1.000
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 4588ms
RDB – Max lat: 938ms
LT – Max lat: 4427ms
mFR – Max lat: 2331ms
All mods – Max lat: 966ms

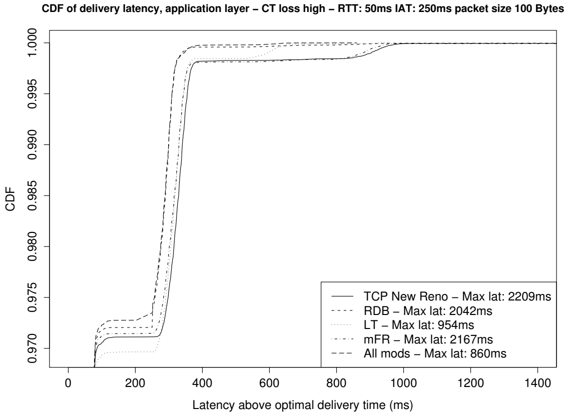

CDF of delivery latency, application layer – CT loss high – RTT: 50ms IAT: 100ms packet size 100 Bytes
CDF
0.93
0.94
0.95
0.96
0.97
0.98
0.99
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 4588ms
RDB – Max lat: 938ms
LT – Max lat: 4427ms
mFR – Max lat: 2331ms
All mods – Max lat: 966ms

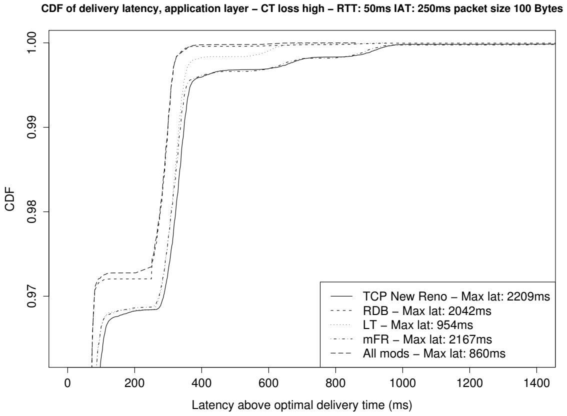

CDF of delivery latency, application layer – CT loss high – RTT: 50ms IAT: 150ms packet size 100 Bytes
CDF
0.970
0.975
0.980
0.985
0.990
0.995
1.000
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 2171ms
RDB – Max lat: 947ms
LT – Max lat: 1827ms
mFR – Max lat: 2142ms
All mods – Max lat: 651ms

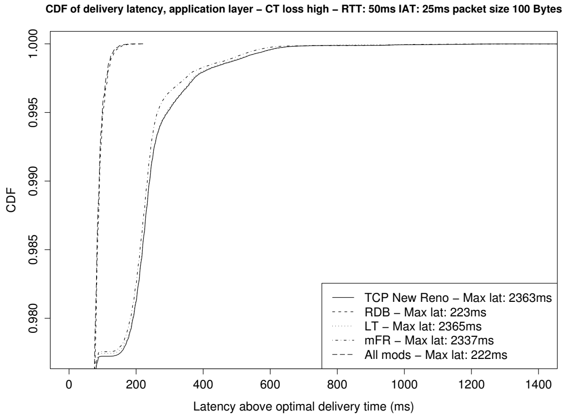

CDF of delivery latency, application layer – CT loss high – RTT: 50ms IAT: 150ms packet size 100 Bytes
CDF
0.95
0.96
0.97
0.98
0.99
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 2171ms
RDB – Max lat: 947ms
LT – Max lat: 1827ms
mFR – Max lat: 2142ms
All mods – Max lat: 651ms

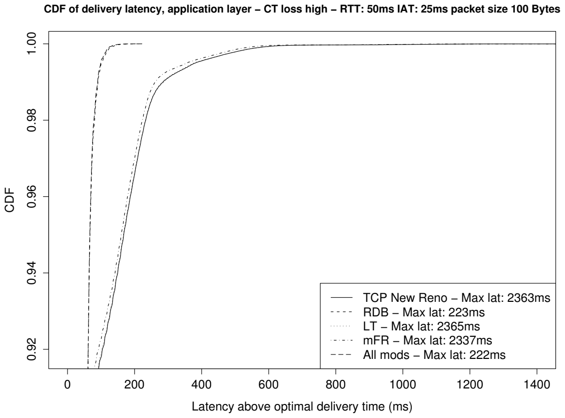

CDF of delivery latency, application layer – CT loss high – RTT: 50ms IAT: 200ms packet size 100 Bytes
CDF
0.970
0.975
0.980
0.985
0.990
0.995
1.000
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 4452ms
RDB – Max lat: 2042ms
LT – Max lat: 1474ms
mFR – Max lat: 4133ms
All mods – Max lat: 1267ms

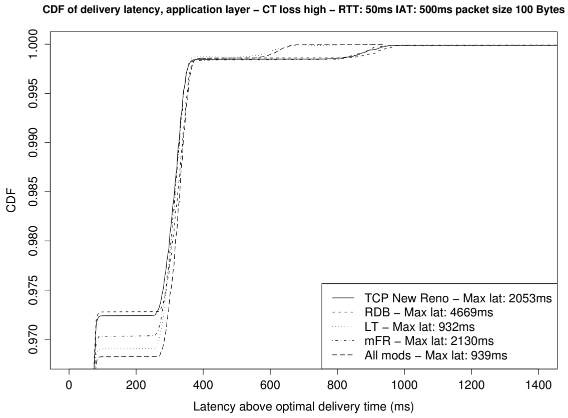

CDF of delivery latency, application layer – CT loss high – RTT: 50ms IAT: 200ms packet size 100 Bytes
CDF
0.95
0.96
0.97
0.98
0.99
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 4452ms
RDB – Max lat: 2042ms
LT – Max lat: 1474ms
mFR – Max lat: 4133ms
All mods – Max lat: 1267ms

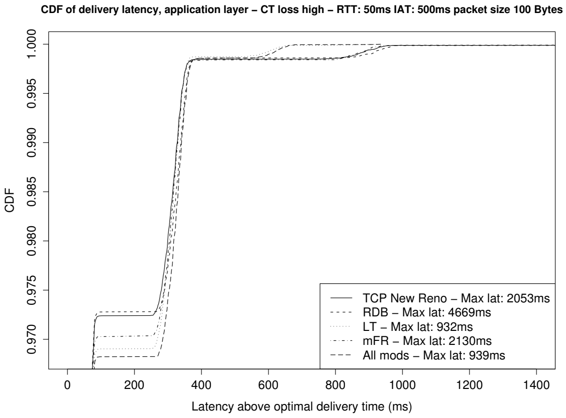

CDF of delivery latency, application layer – CT loss high – RTT: 50ms IAT: 250ms packet size 100 Bytes
CDF
0.970
0.975
0.980
0.985
0.990
0.995
1.000
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 2209ms
RDB – Max lat: 2042ms
LT – Max lat: 954ms
mFR – Max lat: 2167ms
All mods – Max lat: 860ms

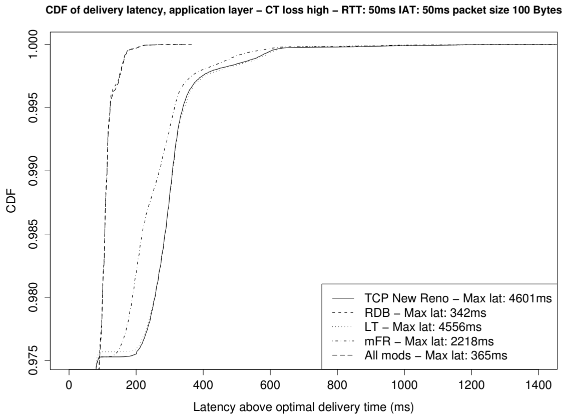

CDF of delivery latency, application layer – CT loss high – RTT: 50ms IAT: 250ms packet size 100 Bytes
CDF
0.97
0.98
0.99
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 2209ms
RDB – Max lat: 2042ms
LT – Max lat: 954ms
mFR – Max lat: 2167ms
All mods – Max lat: 860ms

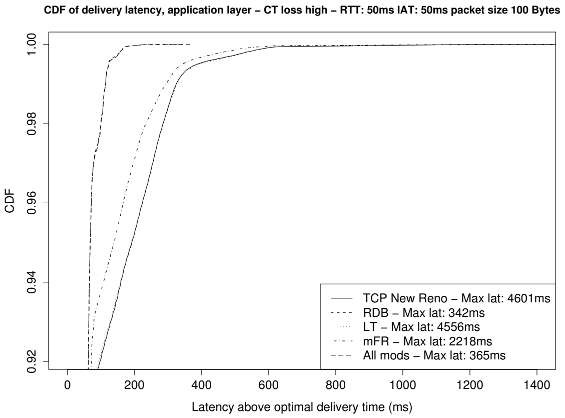

CDF of delivery latency, application layer – CT loss high – RTT: 50ms IAT: 25ms packet size 100 Bytes


CDF of delivery latency, application layer – CT loss high – RTT: 50ms IAT: 25ms packet size 100 Bytes

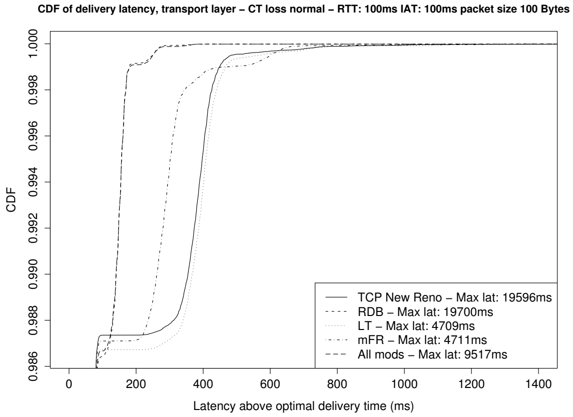

CDF of delivery latency, application layer – CT loss high – RTT: 50ms IAT: 500ms packet size 100 Bytes

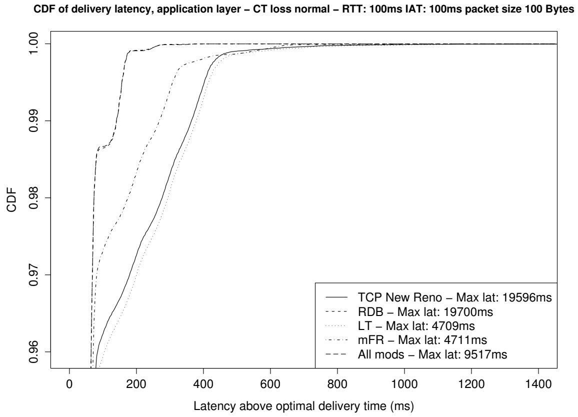

CDF of delivery latency, application layer – CT loss high – RTT: 50ms IAT: 500ms packet size 100 Bytes

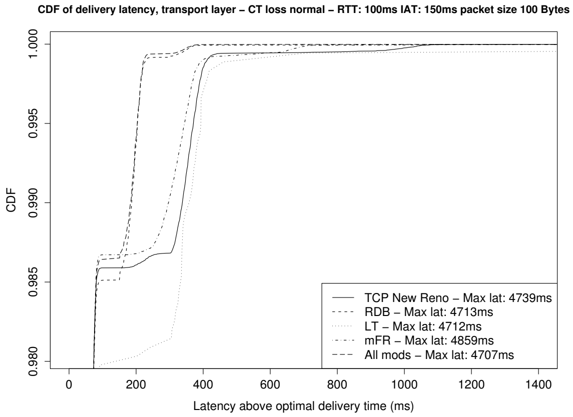

CDF of delivery latency, application layer – CT loss high – RTT: 50ms IAT: 50ms packet size 100 Bytes

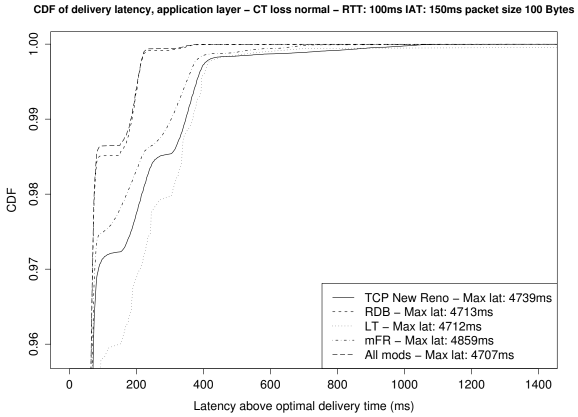

CDF of delivery latency, application layer – CT loss high – RTT: 50ms IAT: 50ms packet size 100 Bytes

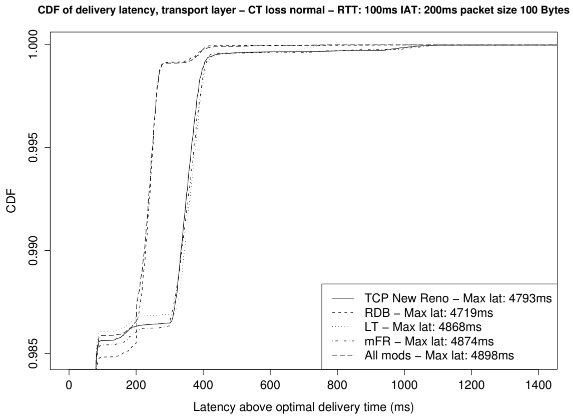
D.3 Cross-traffic loss - low loss rate

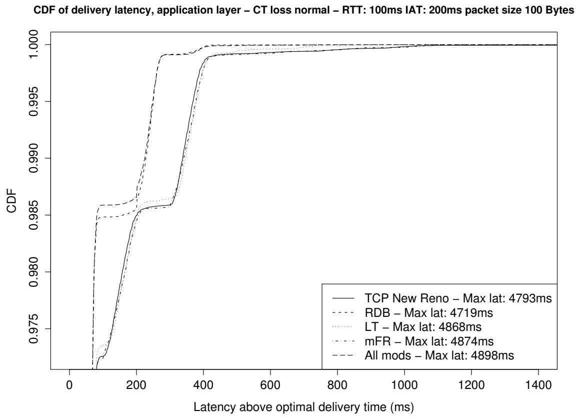

CDF of delivery latency, transport layer – CT loss normal – RTT: 100ms IAT: 100ms packet size 100 Bytes

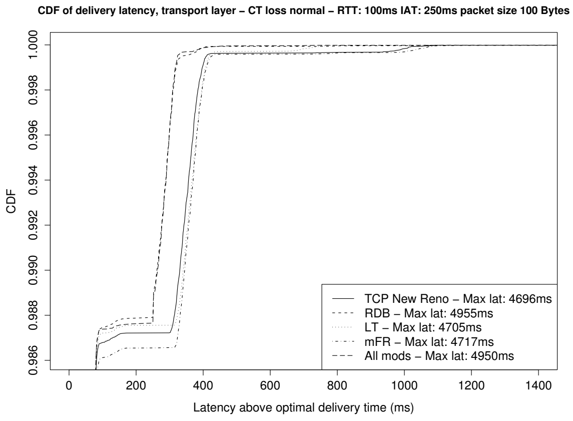

CDF of delivery latency, application layer – CT loss normal – RTT: 100ms IAT: 100ms packet size 100 Bytes

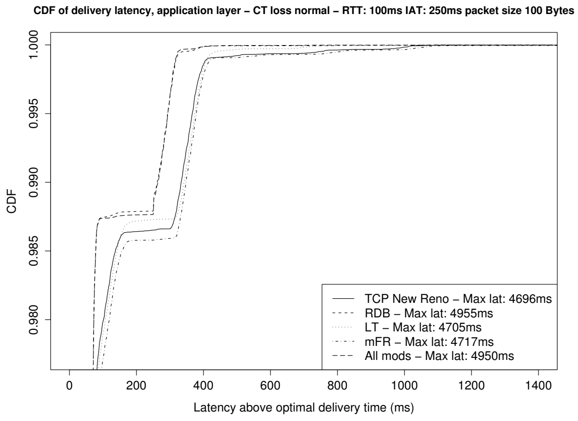

CDF of delivery latency, transport layer – CT loss normal – RTT: 100ms IAT: 150ms packet size 100 Bytes


CDF of delivery latency, application layer – CT loss normal – RTT: 100ms IAT: 150ms packet size 100 Bytes

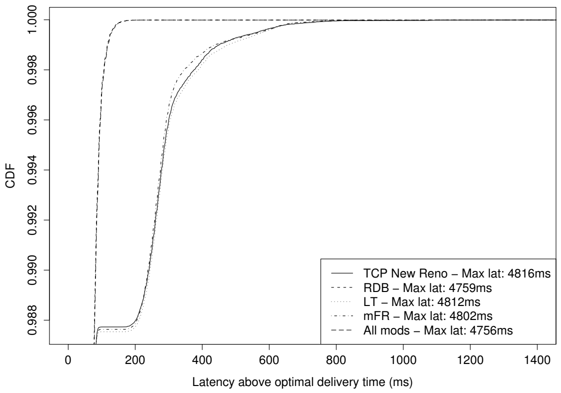

CDF of delivery latency, transport layer – CT loss normal – RTT: 100ms IAT: 200ms packet size 100 Bytes


CDF of delivery latency, application layer – CT loss normal – RTT: 100ms IAT: 200ms packet size 100 Bytes

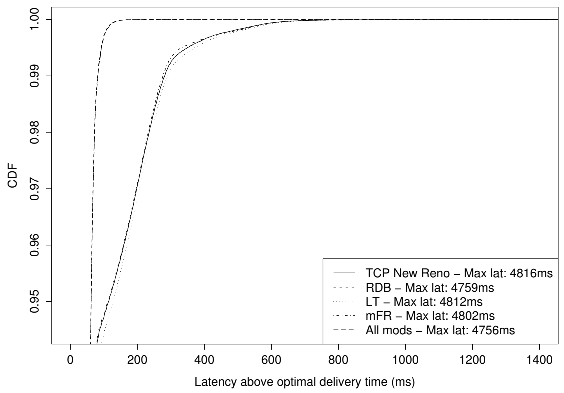

CDF of delivery latency, transport layer – CT loss normal – RTT: 100ms IAT: 250ms packet size 100 Bytes


CDF of delivery latency, application layer – CT loss normal – RTT: 100ms IAT: 250ms packet size 100 Bytes

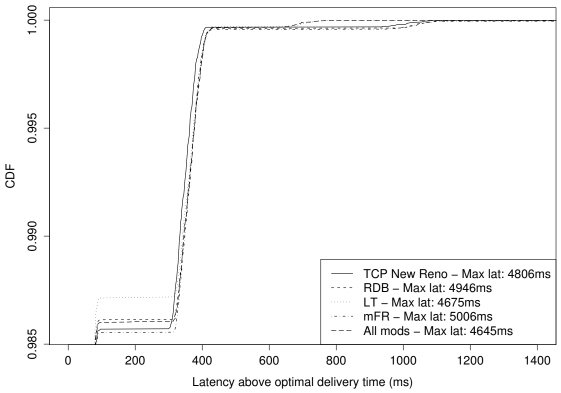
CDF of delivery latency, transport layer – CT loss normal – RTT: 100ms IAT: 25ms packet size 100 Bytes



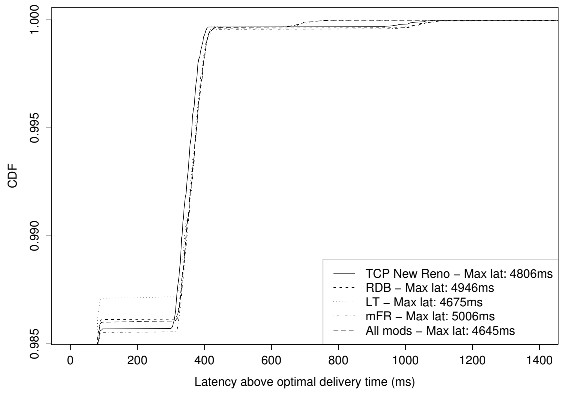
CDF of delivery latency, application layer – CT loss normal – RTT: 100ms IAT: 25ms packet size 100 Bytes



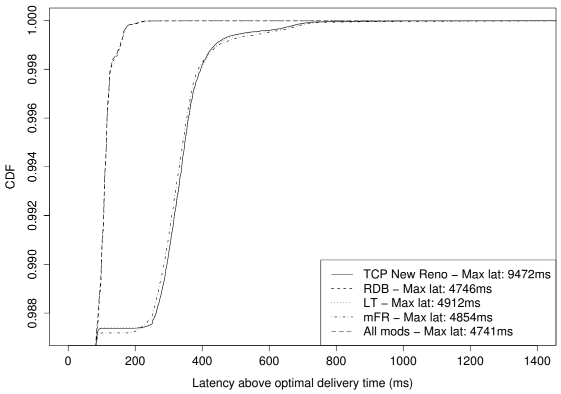
CDF of delivery latency, transport layer – CT loss normal – RTT: 100ms IAT: 500ms packet size 100 Bytes



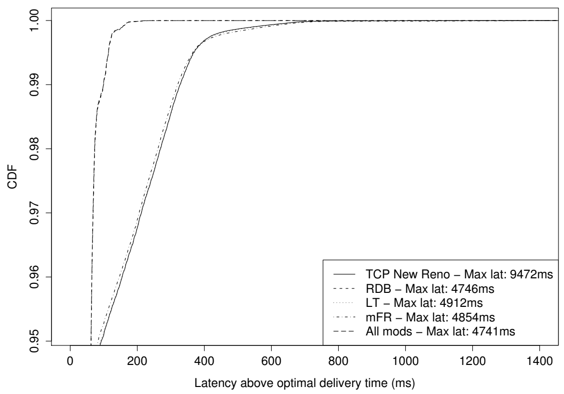
CDF of delivery latency, application layer – CT loss normal – RTT: 100ms IAT: 500ms packet size 100 Bytes



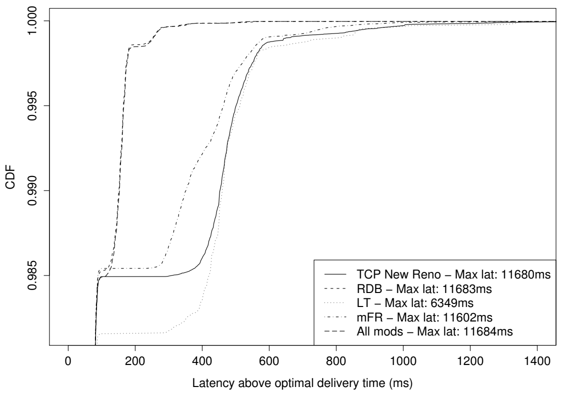
CDF of delivery latency, transport layer – CT loss normal – RTT: 100ms IAT: 50ms packet size 100 Bytes



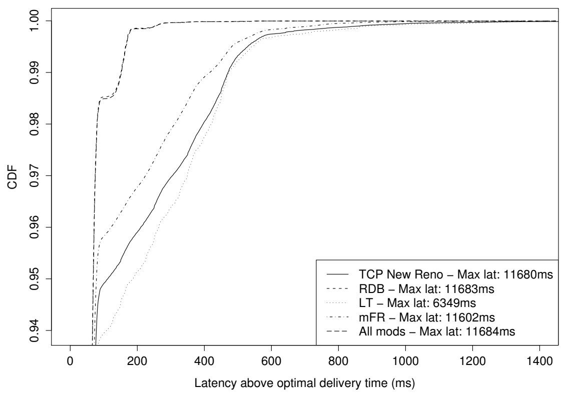
CDF of delivery latency, application layer – CT loss normal – RTT: 100ms IAT: 50ms packet size 100 Bytes



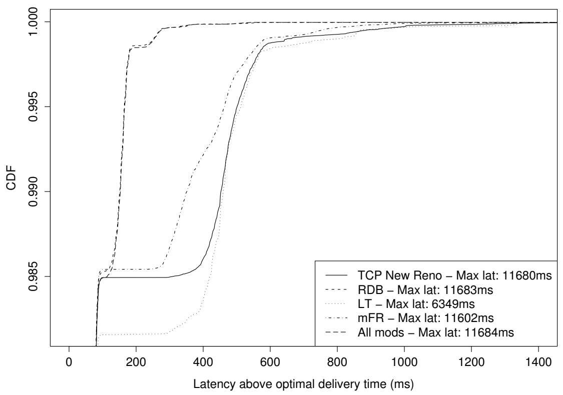
CDF of delivery latency, transport layer – CT loss normal – RTT: 150ms IAT: 100ms packet size 100 Bytes



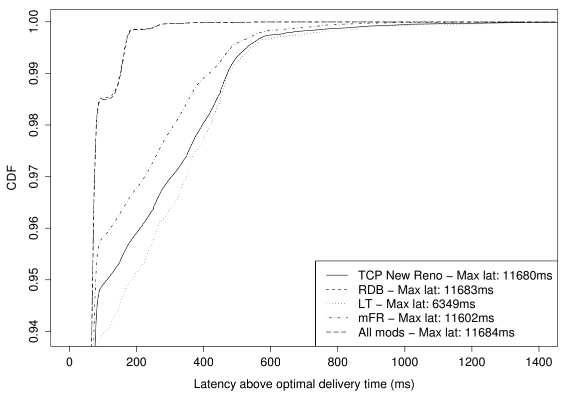
CDF of delivery latency, application layer – CT loss normal – RTT: 150ms IAT: 100ms packet size 100 Bytes



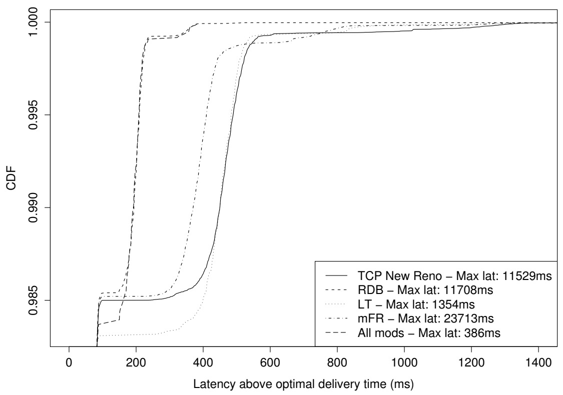
CDF of delivery latency, transport layer – CT loss normal – RTT: 150ms IAT: 100ms packet size 100 Bytes



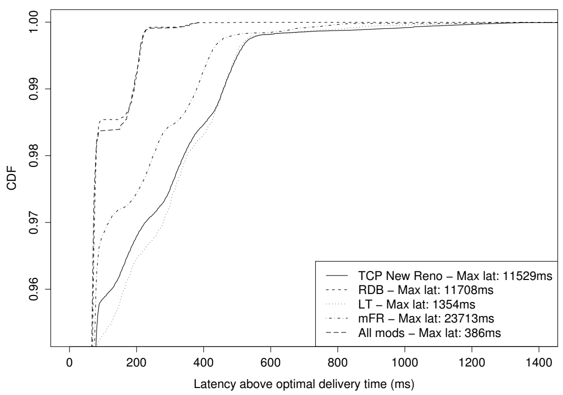
CDF of delivery latency, application layer – CT loss normal – RTT: 150ms IAT: 100ms packet size 100 Bytes



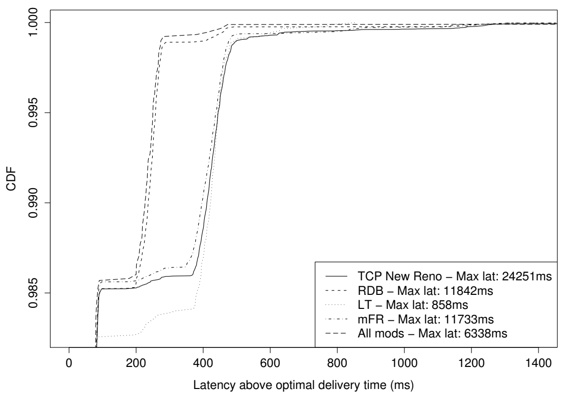
CDF of delivery latency, transport layer – CT loss normal – RTT: 150ms IAT: 150ms packet size 100 Bytes



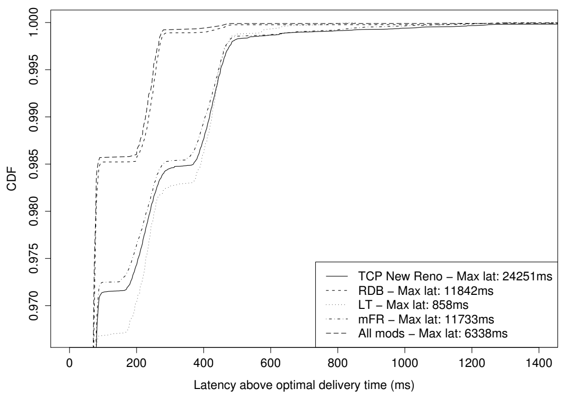
CDF of delivery latency, application layer – CT loss normal – RTT: 150ms IAT: 150ms packet size 100 Bytes



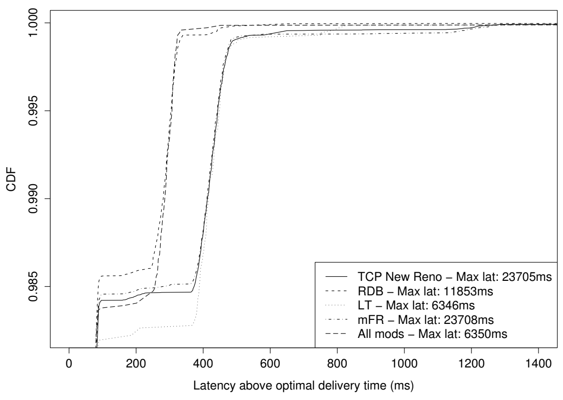
CDF of delivery latency, transport layer – CT loss normal – RTT: 150ms IAT: 200ms packet size 100 Bytes



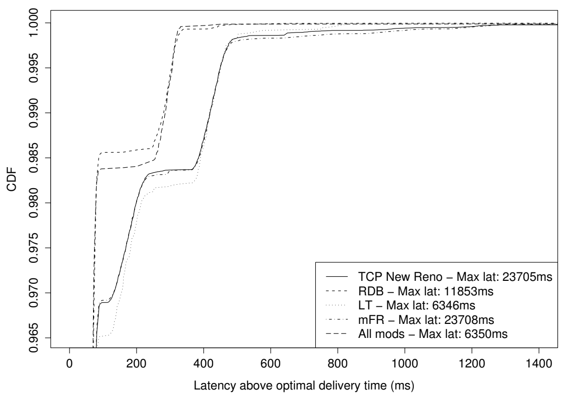
CDF of delivery latency, application layer – CT loss normal – RTT: 150ms IAT: 200ms packet size 100 Bytes

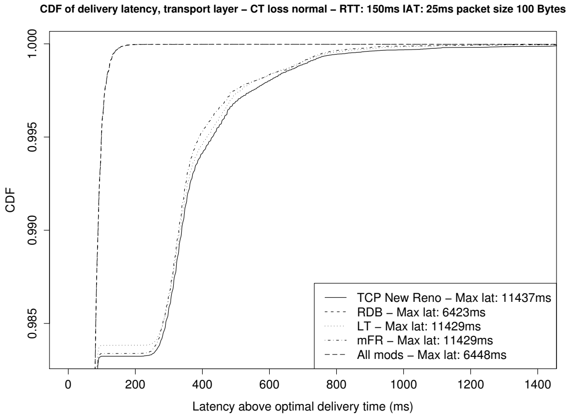


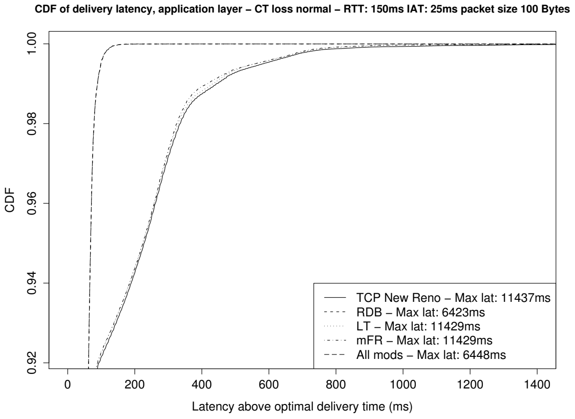
CDF of delivery latency, transport layer – CT loss normal – RTT: 150ms IAT: 250ms packet size 100 Bytes

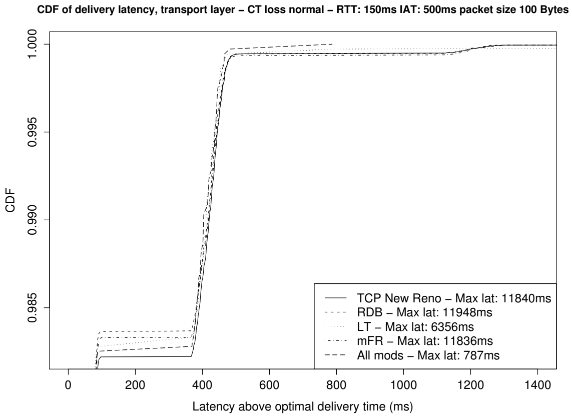


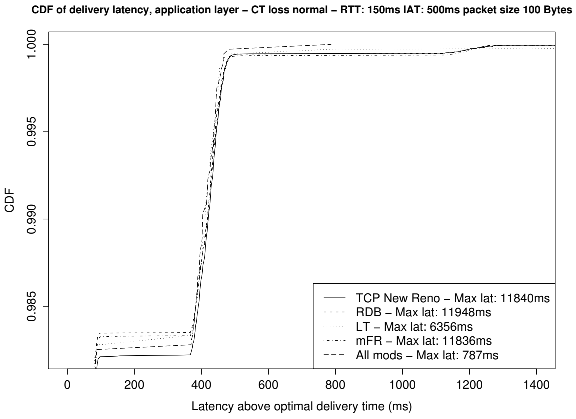
CDF of delivery latency, application layer – CT loss normal – RTT: 150ms IAT: 250ms packet size 100 Bytes

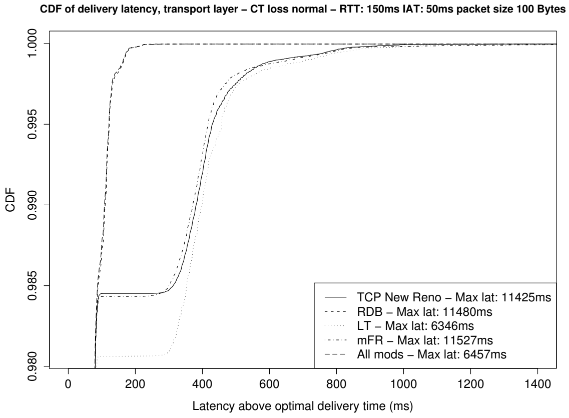


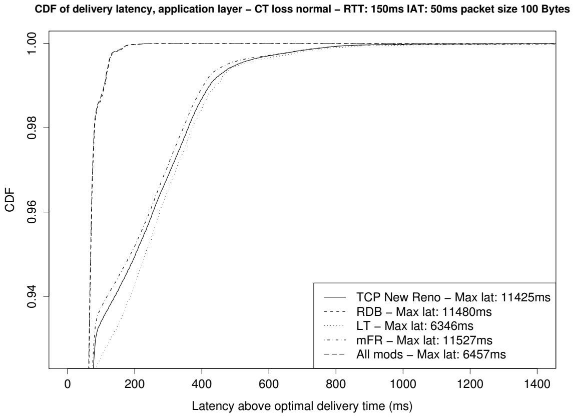

CDF of delivery latency, transport layer – CT loss normal – RTT: 150ms IAT: 25ms packet size 100 Bytes
CDF
0.985
0.990
0.995
1.000
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 11437ms
RDB – Max lat: 6423ms
LT – Max lat: 11429ms
mFR – Max lat: 11429ms
All mods – Max lat: 6448ms

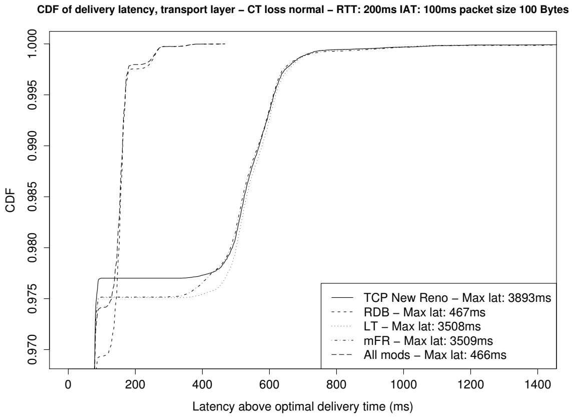

CDF of delivery latency, application layer – CT loss normal – RTT: 150ms IAT: 25ms packet size 100 Bytes
CDF
0.92
0.94
0.96
0.98
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 11437ms
RDB – Max lat: 6423ms
LT – Max lat: 11429ms
mFR – Max lat: 11429ms
All mods – Max lat: 6448ms

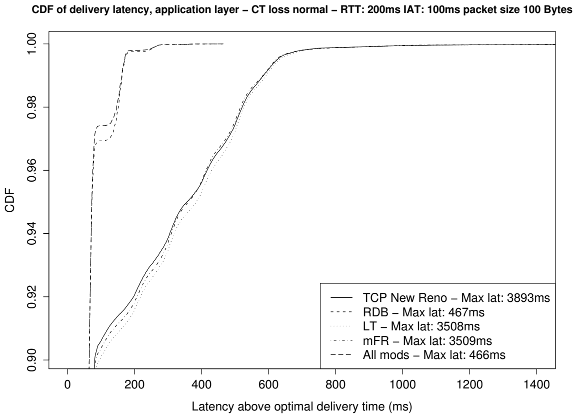

CDF of delivery latency, transport layer – CT loss normal – RTT: 150ms IAT: 500ms packet size 100 Bytes
CDF
0.985
0.990
0.995
1.000
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 11840ms
RDB – Max lat: 11948ms
LT – Max lat: 6356ms
mFR – Max lat: 11836ms
All mods – Max lat: 787ms


CDF of delivery latency, application layer – CT loss normal – RTT: 150ms IAT: 500ms packet size 100 Bytes
CDF
0.985
0.990
0.995
1.000
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 11840ms
RDB – Max lat: 11948ms
LT – Max lat: 6356ms
mFR – Max lat: 11836ms
All mods – Max lat: 787ms

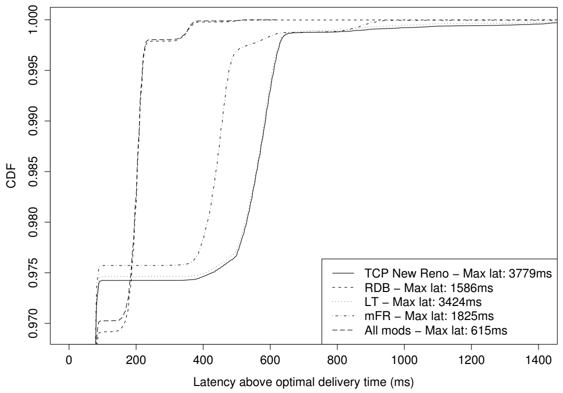

CDF of delivery latency, transport layer – CT loss normal – RTT: 150ms IAT: 50ms packet size 100 Bytes
CDF
0.980
0.985
0.990
0.995
1.000
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 11425ms
RDB – Max lat: 11480ms
LT – Max lat: 6346ms
mFR – Max lat: 11527ms
All mods – Max lat: 6457ms


CDF of delivery latency, application layer – CT loss normal – RTT: 150ms IAT: 50ms packet size 100 Bytes
CDF
0.94
0.96
0.98
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 11425ms
RDB – Max lat: 11480ms
LT – Max lat: 6346ms
mFR – Max lat: 11527ms
All mods – Max lat: 6457ms

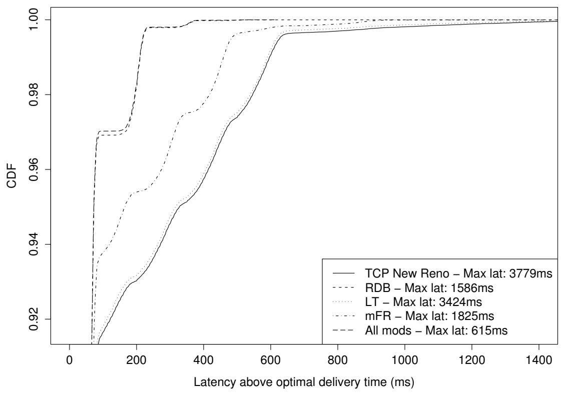

CDF of delivery latency, transport layer – CT loss normal – RTT: 200ms IAT: 100ms packet size 100 Bytes
CDF
0.970
0.975
0.980
0.985
0.990
0.995
1.000
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 3893ms
RDB – Max lat: 467ms
LT – Max lat: 3508ms
mFR – Max lat: 3509ms
All mods – Max lat: 466ms


CDF of delivery latency, application layer – CT loss normal – RTT: 200ms IAT: 100ms packet size 100 Bytes
CDF
0.90
0.92
0.94
0.96
0.98
1.00
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 3893ms
RDB – Max lat: 467ms
LT – Max lat: 3508ms
mFR – Max lat: 3509ms
All mods – Max lat: 466ms

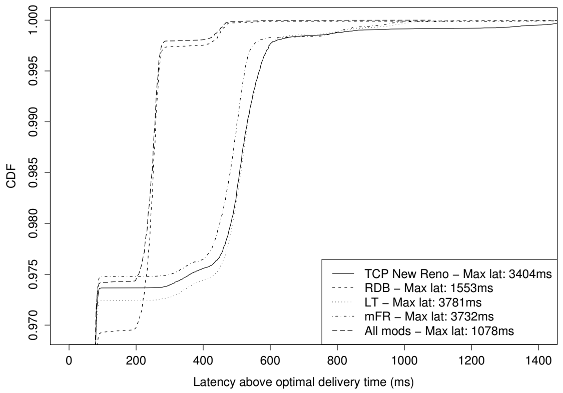
CDF of delivery latency, transport layer – CT loss normal – RTT: 200ms IAT: 150ms packet size 100 Bytes



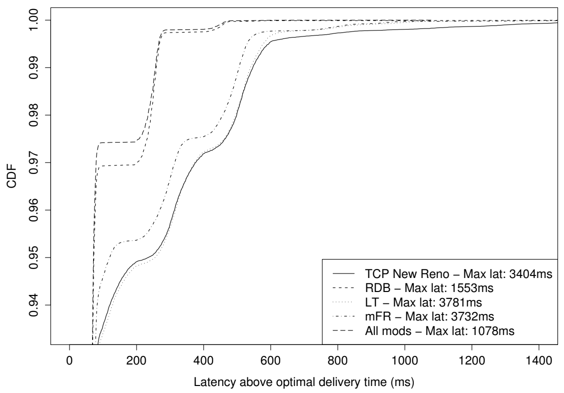
CDF of delivery latency, application layer – CT loss normal – RTT: 200ms IAT: 150ms packet size 100 Bytes



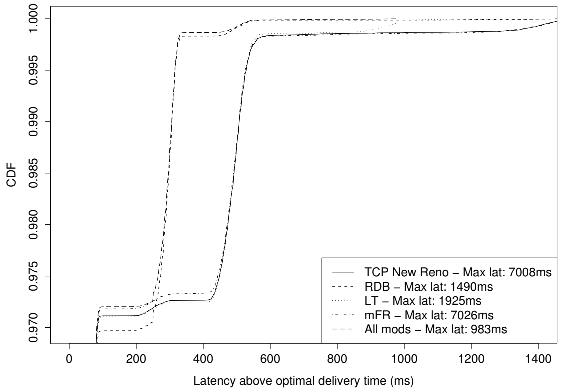
CDF of delivery latency, transport layer – CT loss normal – RTT: 200ms IAT: 200ms packet size 100 Bytes



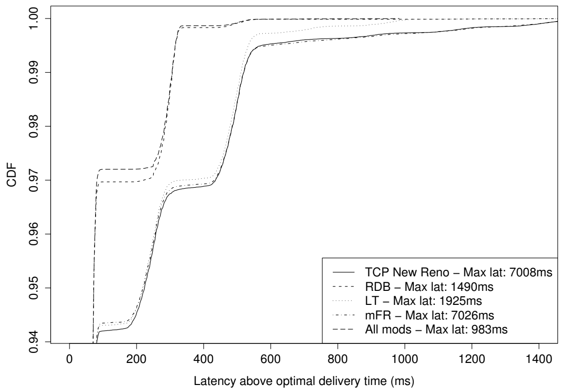
CDF of delivery latency, application layer – CT loss normal – RTT: 200ms IAT: 200ms packet size 100 Bytes



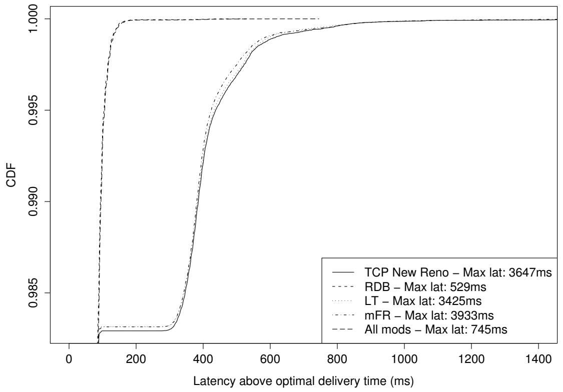
CDF of delivery latency, transport layer – CT loss normal – RTT: 200ms IAT: 250ms packet size 100 Bytes



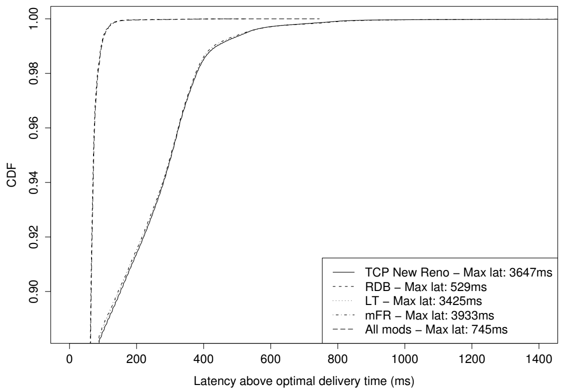
CDF of delivery latency, application layer – CT loss normal – RTT: 200ms IAT: 250ms packet size 100 Bytes



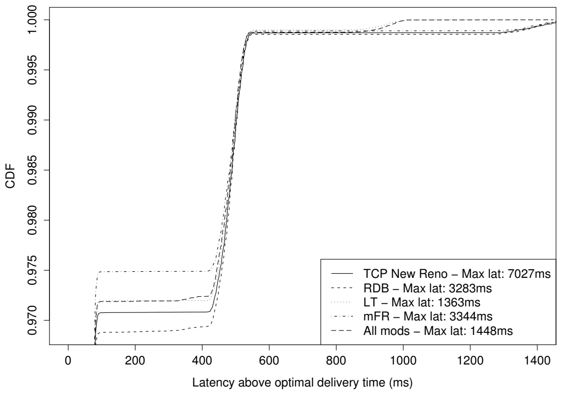
CDF of delivery latency, transport layer – CT loss normal – RTT: 200ms IAT: 25ms packet size 100 Bytes



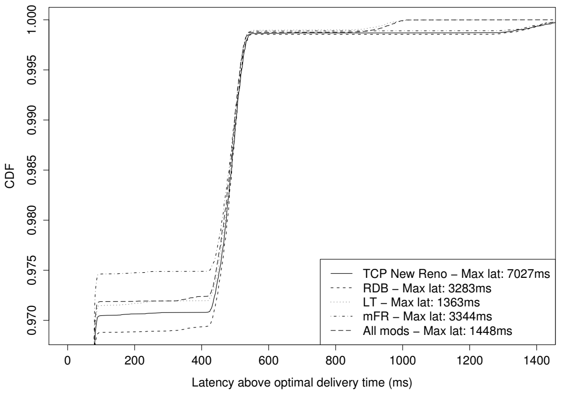
CDF of delivery latency, application layer – CT loss normal – RTT: 200ms IAT: 25ms packet size 100 Bytes



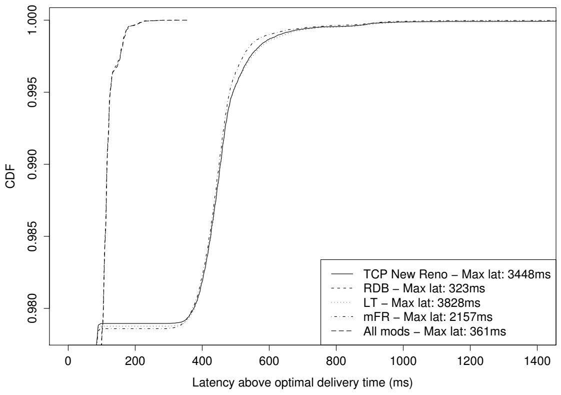
CDF of delivery latency, transport layer – CT loss normal – RTT: 200ms IAT: 500ms packet size 100 Bytes



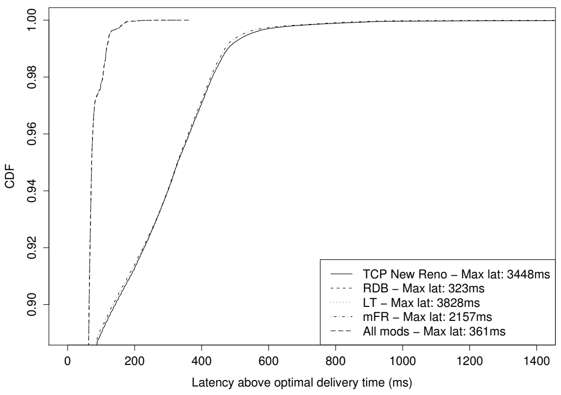
CDF of delivery latency, application layer – CT loss normal – RTT: 200ms IAT: 500ms packet size 100 Bytes



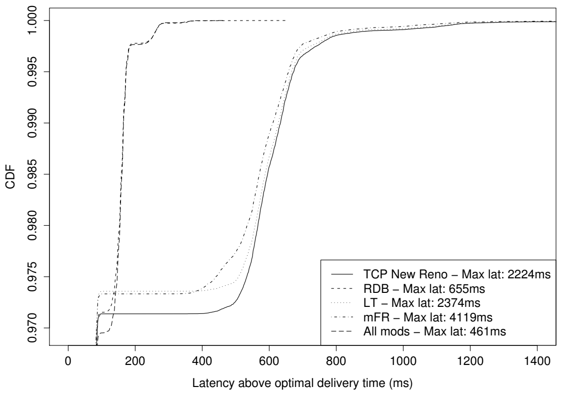
CDF of delivery latency, transport layer – CT loss normal – RTT: 200ms IAT: 50ms packet size 100 Bytes



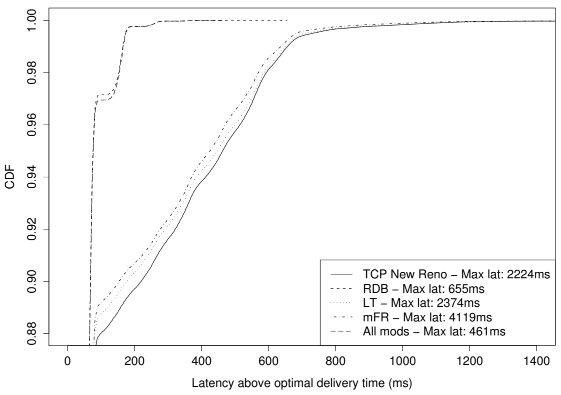
CDF of delivery latency, application layer – CT loss normal – RTT: 200ms IAT: 50ms packet size 100 Bytes



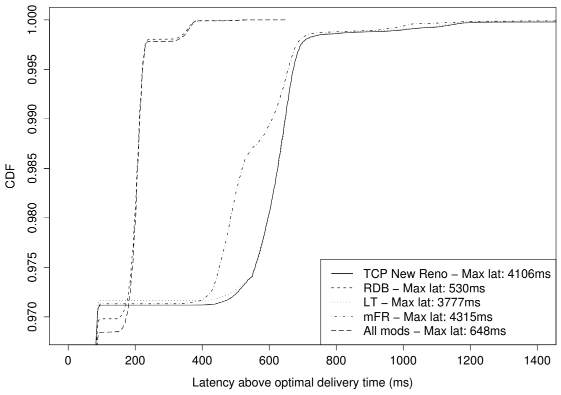
CDF of delivery latency, transport layer – CT loss normal – RTT: 250ms IAT: 100ms packet size 100 Bytes



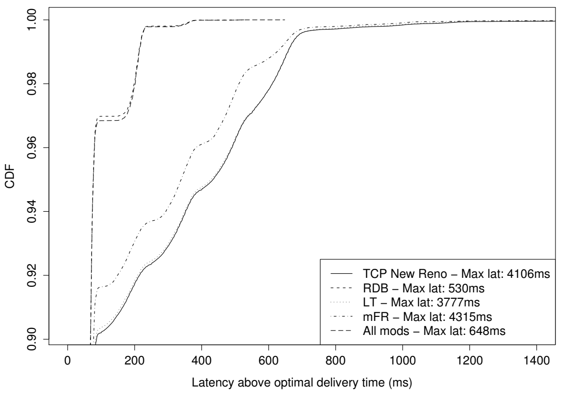
CDF of delivery latency, application layer – CT loss normal – RTT: 250ms IAT: 100ms packet size 100 Bytes



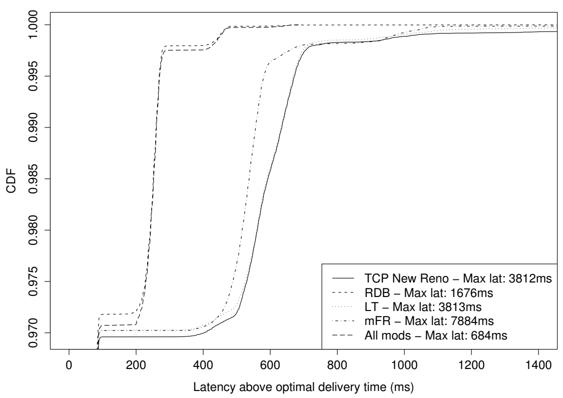
CDF of delivery latency, transport layer – CT loss normal – RTT: 250ms IAT: 150ms packet size 100 Bytes



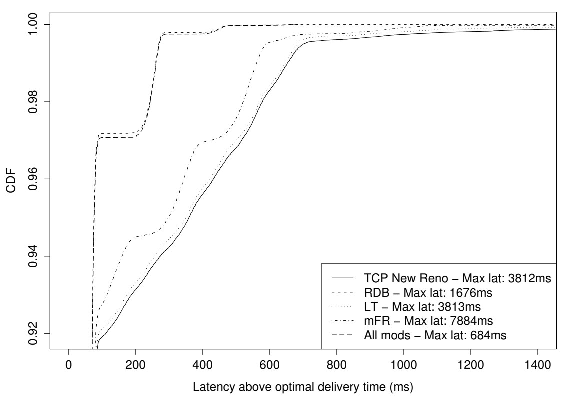
CDF of delivery latency, application layer – CT loss normal – RTT: 250ms IAT: 150ms packet size 100 Bytes



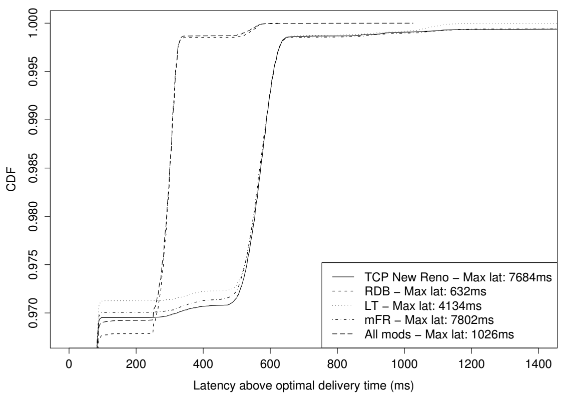
CDF of delivery latency, transport layer – CT loss normal – RTT: 250ms IAT: 200ms packet size 100 Bytes



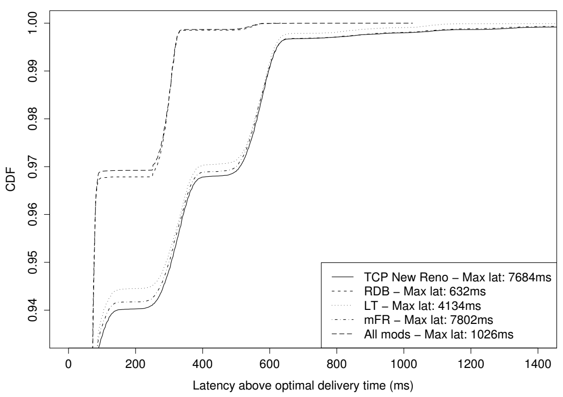
CDF of delivery latency, application layer – CT loss normal – RTT: 250ms IAT: 200ms packet size 100 Bytes



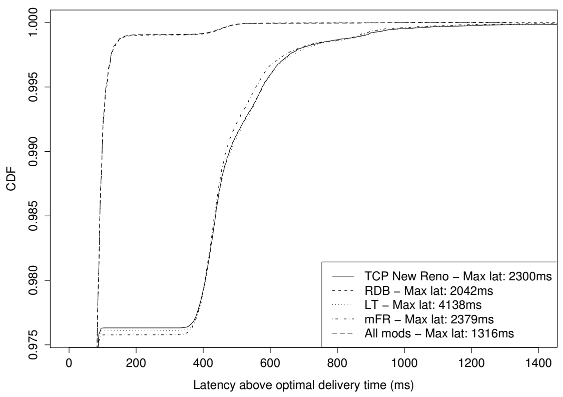
CDF of delivery latency, transport layer – CT loss normal – RTT: 250ms IAT: 250ms packet size 100 Bytes



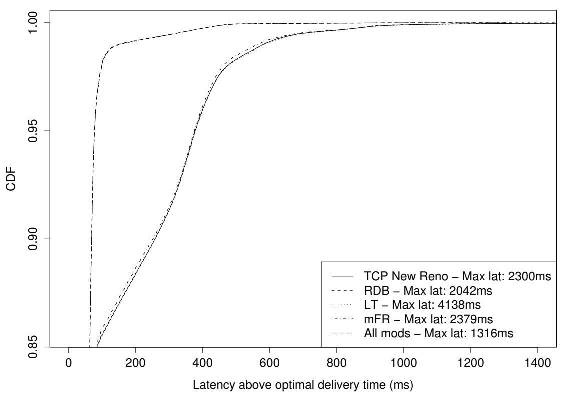
CDF of delivery latency, application layer – CT loss normal – RTT: 250ms IAT: 250ms packet size 100 Bytes



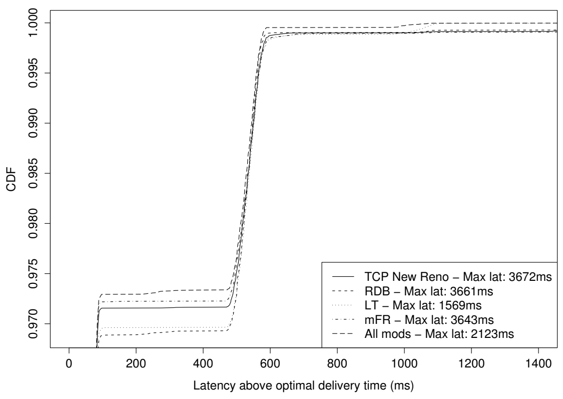
CDF of delivery latency, transport layer – CT loss normal – RTT: 250ms IAT: 25ms packet size 100 Bytes



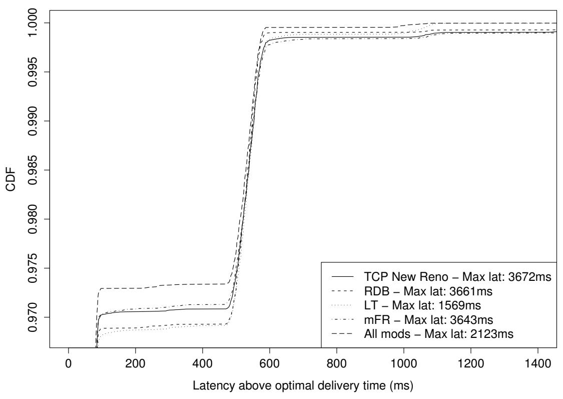
CDF of delivery latency, application layer – CT loss normal – RTT: 250ms IAT: 25ms packet size 100 Bytes



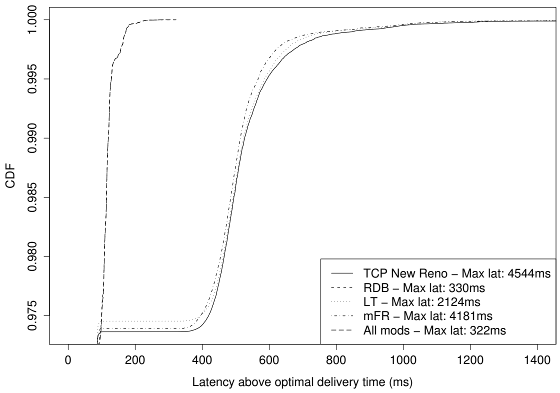
CDF of delivery latency, transport layer – CT loss normal – RTT: 250ms IAT: 500ms packet size 100 Bytes



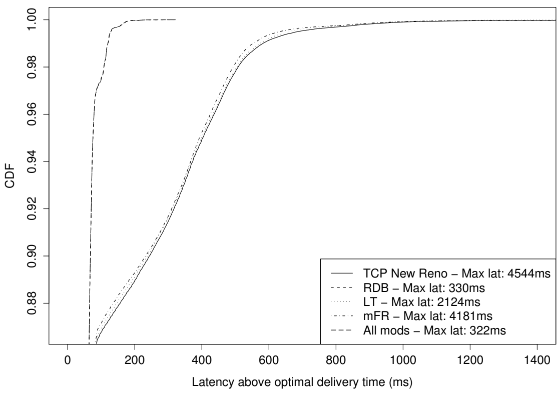
CDF of delivery latency, application layer – CT loss normal – RTT: 250ms IAT: 500ms packet size 100 Bytes



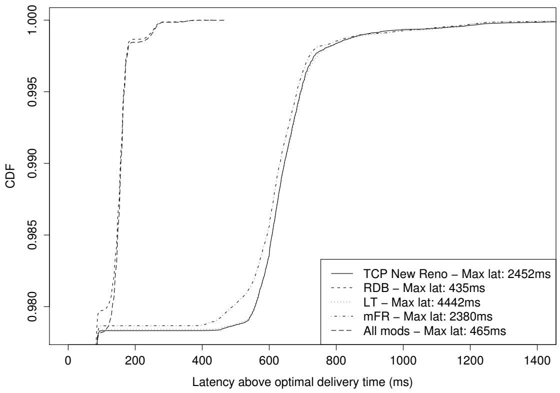
CDF of delivery latency, transport layer – CT loss normal – RTT: 250ms IAT: 50ms packet size 100 Bytes



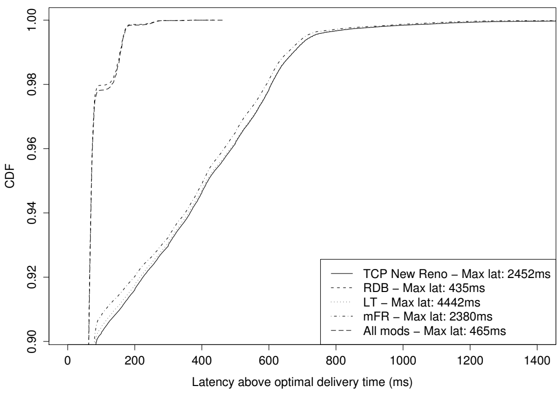
CDF of delivery latency, application layer – CT loss normal – RTT: 250ms IAT: 50ms packet size 100 Bytes



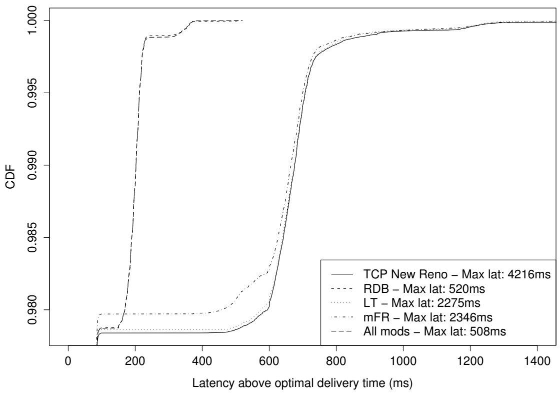
CDF of delivery latency, transport layer – CT loss normal – RTT: 300ms IAT: 100ms packet size 100 Bytes



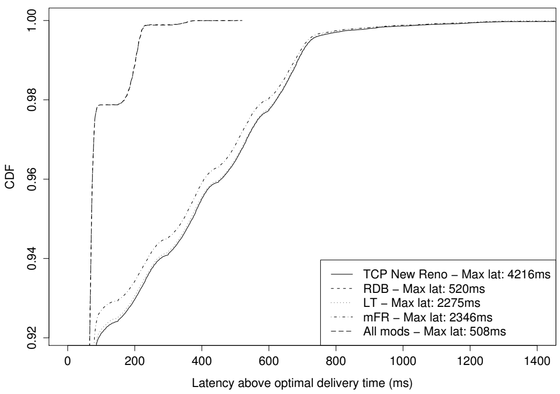
CDF of delivery latency, application layer – CT loss normal – RTT: 300ms IAT: 100ms packet size 100 Bytes



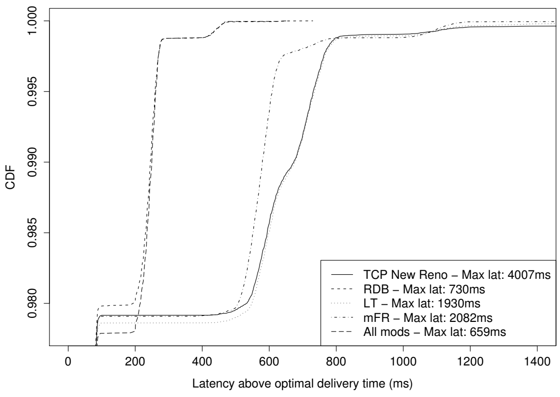
CDF of delivery latency, transport layer – CT loss normal – RTT: 300ms IAT: 150ms packet size 100 Bytes



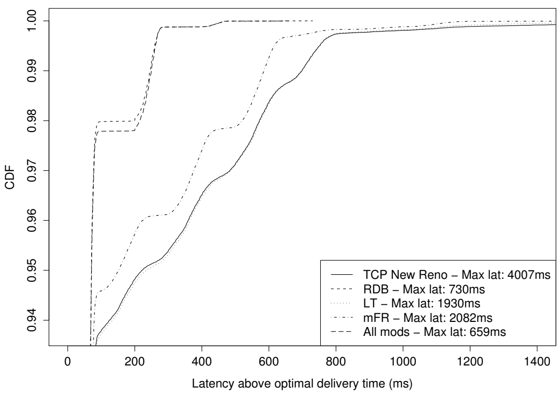
CDF of delivery latency, application layer – CT loss normal – RTT: 300ms IAT: 150ms packet size 100 Bytes

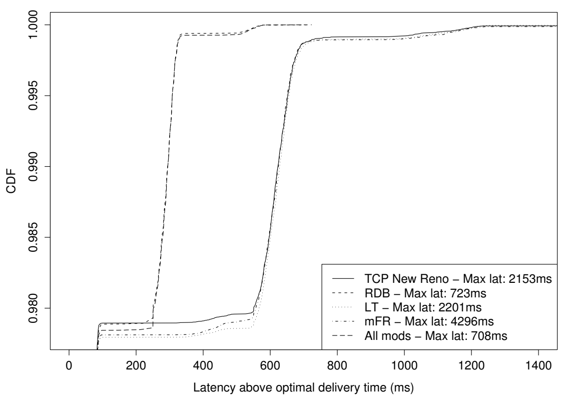


CDF of delivery latency, transport layer – CT loss normal – RTT: 300ms IAT: 200ms packet size 100 Bytes

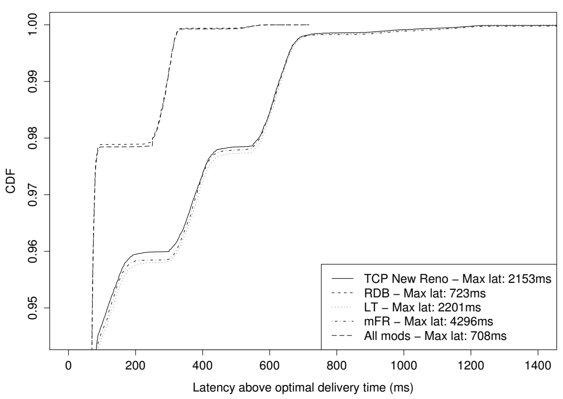


CDF of delivery latency, application layer – CT loss normal – RTT: 300ms IAT: 200ms packet size 100 Bytes

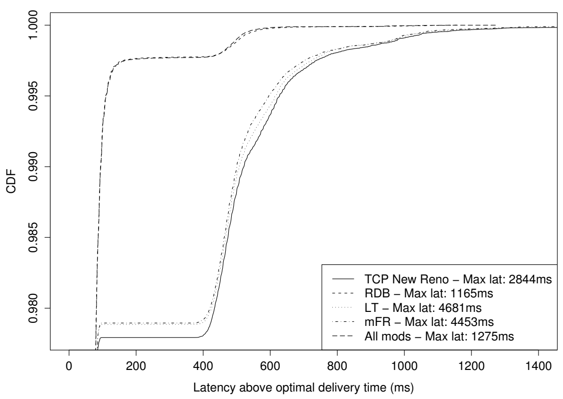




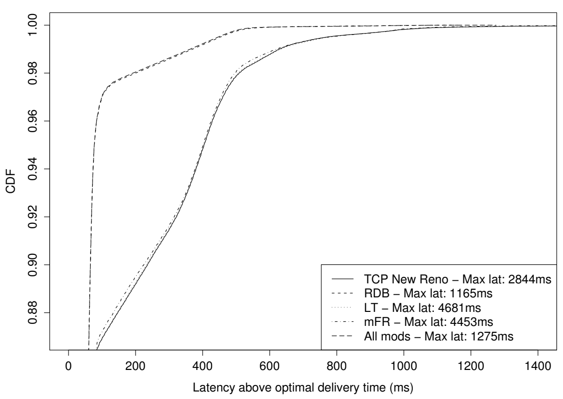
CDF of delivery latency, transport layer – CT loss normal – RTT: 300ms IAT: 250ms packet size 100 Bytes



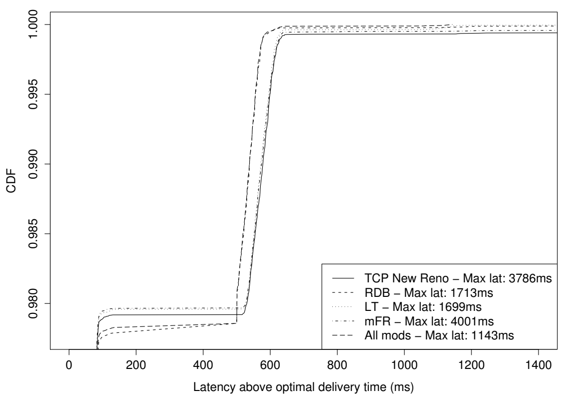
CDF of delivery latency, application layer – CT loss normal – RTT: 300ms IAT: 250ms packet size 100 Bytes



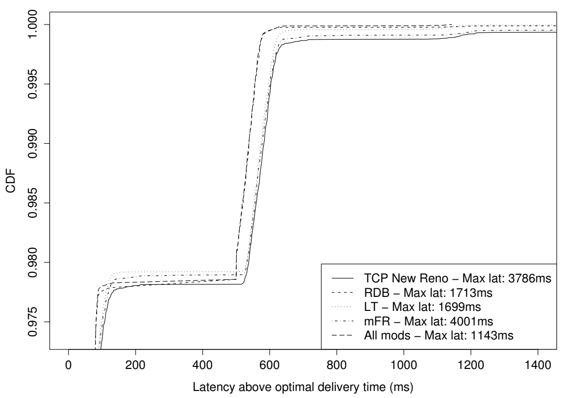
CDF of delivery latency, transport layer – CT loss normal – RTT: 300ms IAT: 25ms packet size 100 Bytes



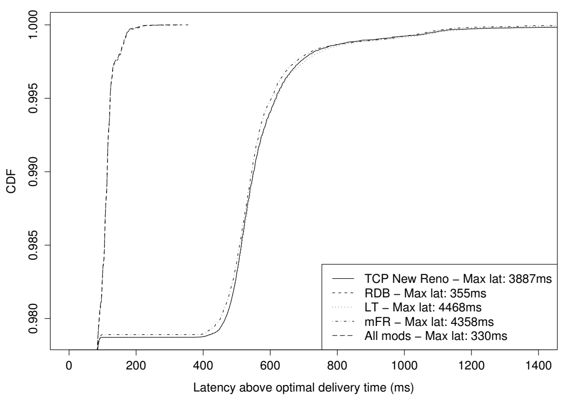
CDF of delivery latency, application layer – CT loss normal – RTT: 300ms IAT: 25ms packet size 100 Bytes



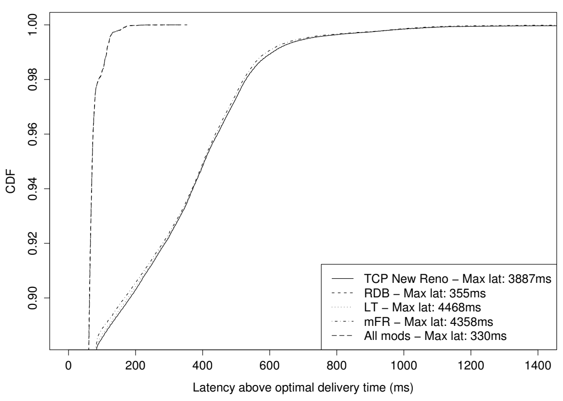
CDF of delivery latency, transport layer – CT loss normal – RTT: 300ms IAT: 500ms packet size 100 Bytes



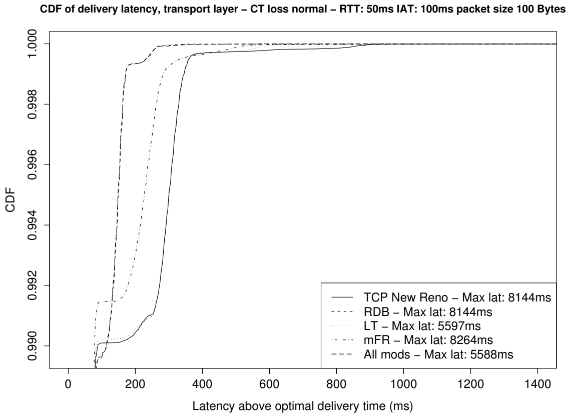
CDF of delivery latency, application layer – CT loss normal – RTT: 300ms IAT: 500ms packet size 100 Bytes

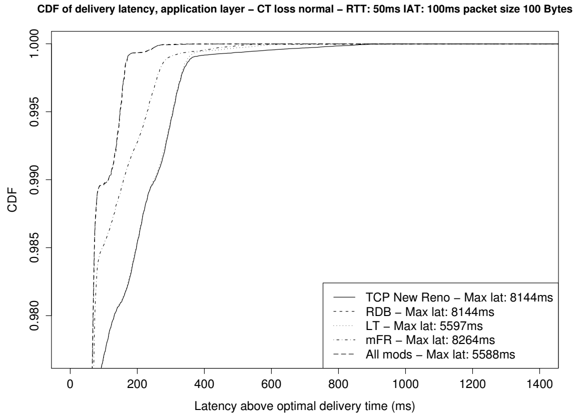


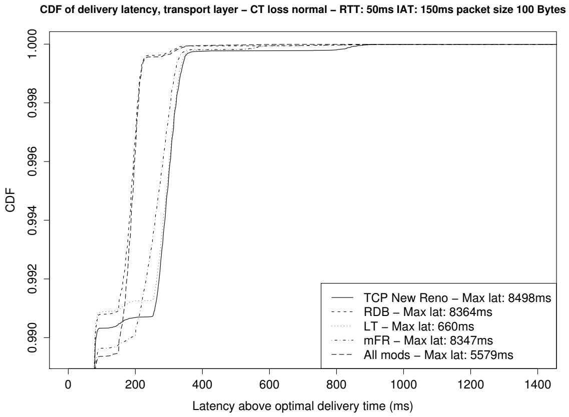
CDF of delivery latency, transport layer – CT loss normal – RTT: 300ms IAT: 50ms packet size 100 Bytes

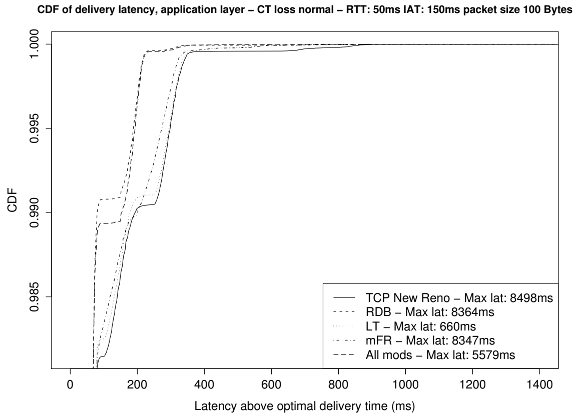


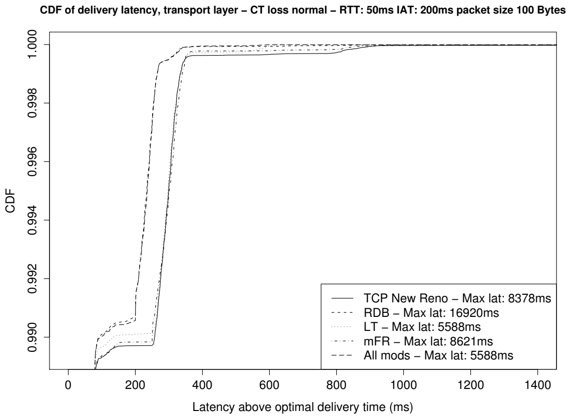
CDF of delivery latency, application layer – CT loss normal – RTT: 300ms IAT: 50ms packet size 100 Bytes

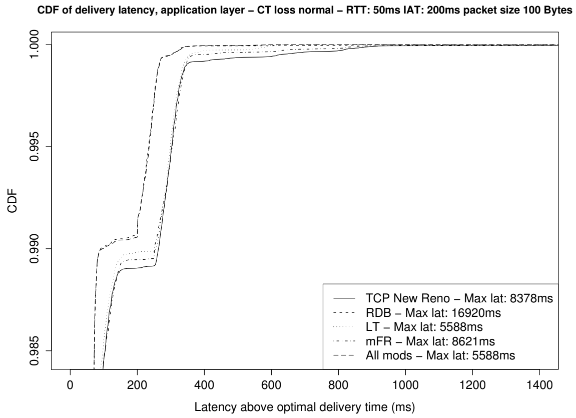

CDF of delivery latency, transport layer – CT loss normal – RTT: 50ms IAT: 100ms packet size 100 Bytes

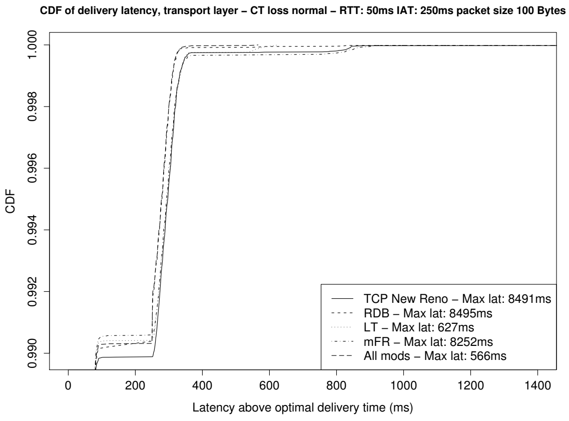

CDF of delivery latency, application layer – CT loss normal – RTT: 50ms IAT: 100ms packet size 100 Bytes

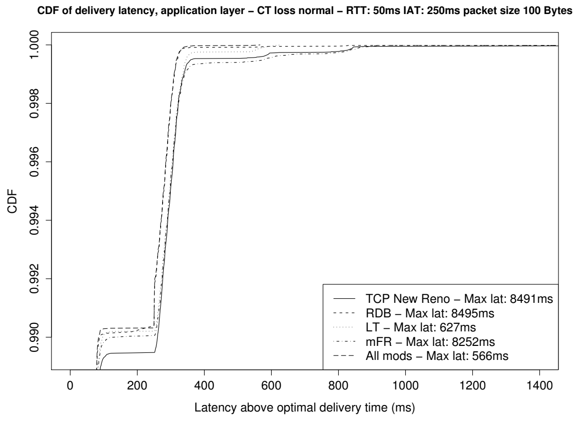

CDF of delivery latency, transport layer – CT loss normal – RTT: 50ms IAT: 150ms packet size 100 Bytes

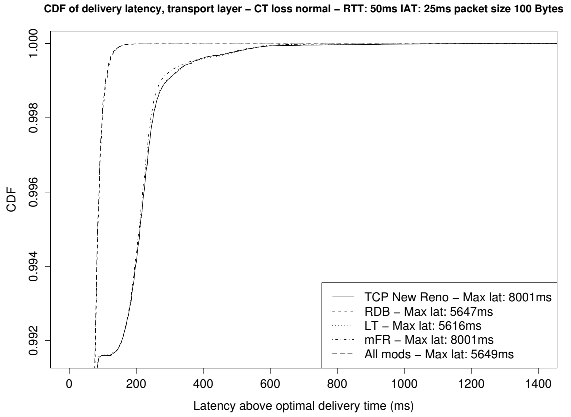

CDF of delivery latency, application layer – CT loss normal – RTT: 50ms IAT: 150ms packet size 100 Bytes

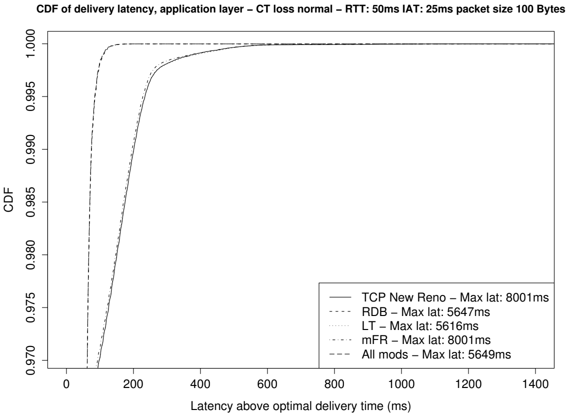

CDF of delivery latency, transport layer – CT loss normal – RTT: 50ms IAT: 200ms packet size 100 Bytes

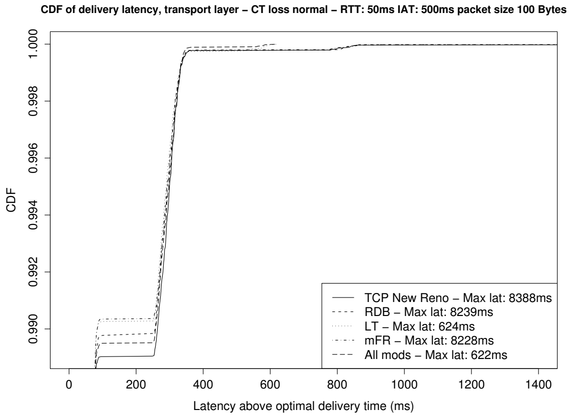

CDF of delivery latency, application layer – CT loss normal – RTT: 50ms IAT: 200ms packet size 100 Bytes

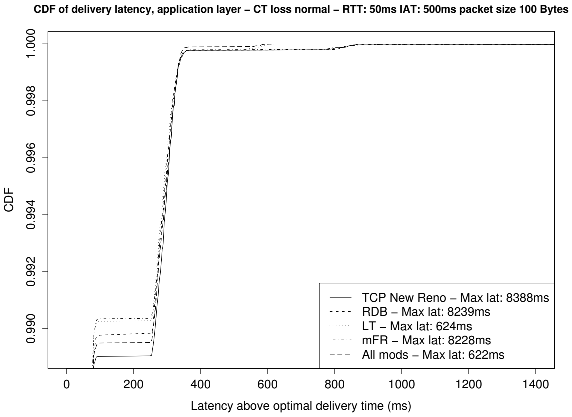

CDF of delivery latency, transport layer – CT loss normal – RTT: 50ms IAT: 250ms packet size 100 Bytes

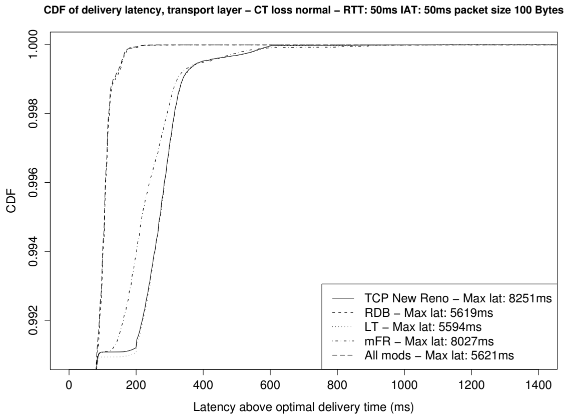

CDF of delivery latency, application layer – CT loss normal – RTT: 50ms IAT: 250ms packet size 100 Bytes

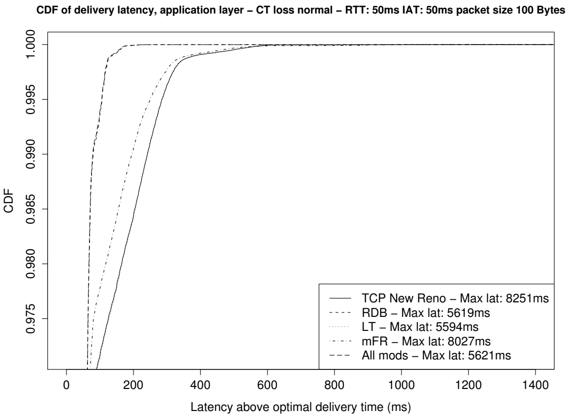

CDF of delivery latency, transport layer – CT loss normal – RTT: 50ms IAT: 25ms packet size 100 Bytes
CDF
0.992
0.994
0.996
0.998
1.000
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 8001ms
RDB – Max lat: 5647ms
LT – Max lat: 5616ms
mFR – Max lat: 8001ms
All mods – Max lat: 5649ms


CDF of delivery latency, application layer – CT loss normal – RTT: 50ms IAT: 25ms packet size 100 Bytes
CDF
0.970
0.975
0.980
0.985
0.990
0.995
1.000
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 8001ms
RDB – Max lat: 5647ms
LT – Max lat: 5616ms
mFR – Max lat: 8001ms
All mods – Max lat: 5649ms


CDF of delivery latency, transport layer – CT loss normal – RTT: 50ms IAT: 500ms packet size 100 Bytes
CDF
0.990
0.992
0.994
0.996
0.998
1.000
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 8388ms
RDB – Max lat: 8239ms
LT – Max lat: 624ms
mFR – Max lat: 8228ms
All mods – Max lat: 622ms


CDF of delivery latency, application layer – CT loss normal – RTT: 50ms IAT: 500ms packet size 100 Bytes
CDF
0.990
0.992
0.994
0.996
0.998
1.000
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 8388ms
RDB – Max lat: 8239ms
LT – Max lat: 624ms
mFR – Max lat: 8228ms
All mods – Max lat: 622ms


CDF of delivery latency, transport layer – CT loss normal – RTT: 50ms IAT: 50ms packet size 100 Bytes
CDF
0.992
0.994
0.996
0.998
1.000
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 8251ms
RDB – Max lat: 5619ms
LT – Max lat: 5594ms
mFR – Max lat: 8027ms
All mods – Max lat: 5621ms


CDF of delivery latency, application layer – CT loss normal – RTT: 50ms IAT: 50ms packet size 100 Bytes
CDF
0.975
0.980
0.985
0.990
0.995
1.000
0
200
400
600
800
1000
1200
1400
Latency above optimal delivery time (ms)
TCP New Reno – Max lat: 8251ms
RDB – Max lat: 5619ms
LT – Max lat: 5594ms
mFR – Max lat: 8027ms
All mods – Max lat: 5621ms

Appendix E

TCP-patch for Linux 2.6.23 kernel

```
diff --git a/include/linux/sysctl.h b/include/linux/sysctl.h
index 483050c..f0edacd 100644
--- a/include/linux/sysctl.h
+++ b/include/linux/sysctl.h
@@ -355,6 +355,11 @@ enum
        NET_IPV4_ROUTE=18,
        NET_IPV4_FIB_HASH=19,
        NET_IPV4_NETFILTER=20,
+
+       NET_IPV4_TCP_FORCE_THIN_RDB=29,         /* Added @ Simula */
+       NET_IPV4_TCP_FORCE_THIN_RM_EXPB=30,     /* Added @ Simula */
+       NET_IPV4_TCP_FORCE_THIN_DUPACK=31,      /* Added @ Simula */
+       NET_IPV4_TCP_RDB_MAX_BUNDLE_BYTES=32,   /* Added @ Simula */

        NET_IPV4_TCP_TIMESTAMPS=33,
        NET_IPV4_TCP_WINDOW_SCALING=34,
diff --git a/include/linux/tcp.h b/include/linux/tcp.h
index c6b9f92..c11a564 100644
--- a/include/linux/tcp.h
+++ b/include/linux/tcp.h
@@ -97,6 +97,10 @@ enum {
 #define TCP_CONGESTION         13      /* Congestion control algorithm */
 #define TCP_MD5SIG             14      /* TCP MD5 Signature (RFC2385) */

+#define TCP_THIN_RDB            15      /* Added @ Simula - Enable redundant data bundling  */
+#define TCP_THIN_RM_EXPB        16      /* Added @ Simula - Remove exponential backoff  */
+#define TCP_THIN_DUPACK         17      /* Added @ Simula - Reduce number of dupAcks needed */
+
 #define TCPI_OPT_TIMESTAMPS    1
 #define TCPI_OPT_SACK          2
 #define TCPI_OPT_WSCALE                4
@@ -296,6 +300,10 @@ struct tcp_sock {
        u8      nonagle;        /* Disable Nagle algorithm?             */
        u8      keepalive_probes; /* num of allowed keep alive probes   */

+       u8      thin_rdb;       /* Enable RDB                           */
+       u8      thin_rm_expb;   /* Remove exp. backoff                  */
+       u8      thin_dupack;    /* Remove dupack                        */
+
 /* RTT measurement */
        u32     srtt;           /* smoothed round trip time << 3        */
        u32     mdev;           /* medium deviation                 */
diff --git a/include/net/sock.h b/include/net/sock.h
index dfeb8b1..af831d1 100644
--- a/include/net/sock.h
+++ b/include/net/sock.h
@@ -462,7 +462,10 @@ static inline void sk_stream_set_owner_r(struct sk_buff *skb, struct sock *sk)

 static inline void sk_stream_free_skb(struct sock *sk, struct sk_buff *skb)
 {
-       skb_truesize_check(skb);
+       /* Modified @ Simula
+          skb_truesize_check creates unnecessary
+          noise when combined with RDB */
+       //skb_truesize_check(skb);
        sock_set_flag(sk, SOCK_QUEUE_SHRUNK);
        sk->sk_wmem_queued   -= skb->truesize;
        sk->sk_forward_alloc += skb->truesize;
diff --git a/include/net/tcp.h b/include/net/tcp.h
index 54053de..411cc9b 100644
--- a/include/net/tcp.h
+++ b/include/net/tcp.h
@@ -188,9 +188,19 @@ extern void tcp_time_wait(struct sock *sk, int state, int timeo);
 #define TCP_NAGLE_CORK         2       /* Socket is corked         */
 #define TCP_NAGLE_PUSH         4       /* Cork is overridden for already queued data */

+/* Added @ Simula - Thin stream support */
+#define TCP_FORCE_THIN_RDB            0 /* Thin streams: exp. backoff   default off */
+#define TCP_FORCE_THIN_RM_EXPB        0 /* Thin streams: dynamic dupack default off */
+#define TCP_FORCE_THIN_DUPACK         0 /* Thin streams: smaller minRTO default off */
+#define TCP_RDB_MAX_BUNDLE_BYTES      0 /* Thin streams: Limit maximum bundled bytes */
+
 extern struct inet_timewait_death_row tcp_death_row;

 /* sysctl variables for tcp */
+extern int sysctl_tcp_force_thin_rdb;          /* Added @ Simula */
+extern int sysctl_tcp_force_thin_rm_expb;      /* Added @ Simula */
+extern int sysctl_tcp_force_thin_dupack;       /* Added @ Simula */
+extern int sysctl_tcp_rdb_max_bundle_bytes;    /* Added @ Simula */
 extern int sysctl_tcp_timestamps;
 extern int sysctl_tcp_window_scaling;
 extern int sysctl_tcp_sack;
@@ -723,6 +733,16 @@ static inline unsigned int tcp_packets_in_flight(const struct tcp_sock *tp)
        return (tp->packets_out - tp->left_out + tp->retrans_out);
 }

+/* Added @ Simula
+ *
+ * To determine whether a stream is thin or not
+ * return 1 if thin, 0 othervice
+ */
+static inline unsigned int tcp_stream_is_thin(const struct tcp_sock *tp)
+{
+       return (tp->packets_out < 4 ? 1 : 0);
+}
+
 /* If cwnd > ssthresh, we may raise ssthresh to be half-way to cwnd.
  * The exception is rate halving phase, when cwnd is decreasing towards
  * ssthresh.
diff --git a/net/ipv4/sysctl_net_ipv4.c b/net/ipv4/sysctl_net_ipv4.c
```

```
index 53ef0f4..58ac82b 100644
--- a/net/ipv4/sysctl_net_ipv4.c
+++ b/net/ipv4/sysctl_net_ipv4.c
@@ -187,6 +187,38 @@ static int strategy_allowed_congestion_control(ctl_table *table, int __user *nam
 }

 ctl_table ipv4_table[] = {
+       {       /* Added @ Simula for thin streams */
+               .ctl_name       = NET_IPV4_TCP_FORCE_THIN_RDB,
+               .procname       = "tcp_force_thin_rdb",
+               .data           = &sysctl_tcp_force_thin_rdb,
+               .maxlen         = sizeof(int),
+               .mode           = 0644,
+               .proc_handler   = &proc_dointvec
+       },
+       {       /* Added @ Simula for thin streams */
+               .ctl_name       = NET_IPV4_TCP_FORCE_THIN_RM_EXPB,
+               .procname       = "tcp_force_thin_rm_expb",
+               .data           = &sysctl_tcp_force_thin_rm_expb,
+               .maxlen         = sizeof(int),
+               .mode           = 0644,
+               .proc_handler   = &proc_dointvec
+       },
+       {       /* Added @ Simula for thin streams */
+               .ctl_name       = NET_IPV4_TCP_FORCE_THIN_DUPACK,
+               .procname       = "tcp_force_thin_dupack",
+               .data           = &sysctl_tcp_force_thin_dupack,
+               .maxlen         = sizeof(int),
+               .mode           = 0644,
+               .proc_handler   = &proc_dointvec
+       },
+       {       /* Added @ Simula for thin streams */
+               .ctl_name       = NET_IPV4_TCP_RDB_MAX_BUNDLE_BYTES,
+               .procname       = "tcp_rdb_max_bundle_bytes",
+               .data           = &sysctl_tcp_rdb_max_bundle_bytes,
+               .maxlen         = sizeof(int),
+               .mode           = 0644,
+               .proc_handler   = &proc_dointvec
+       },
        {
                .ctl_name       = NET_IPV4_TCP_TIMESTAMPS,
                .procname       = "tcp_timestamps",
diff --git a/net/ipv4/tcp.c b/net/ipv4/tcp.c
index 7e74011..8aeec1b 100644
--- a/net/ipv4/tcp.c
+++ b/net/ipv4/tcp.c
@@ -270,6 +270,10 @@

 int sysctl_tcp_fin_timeout __read_mostly = TCP_FIN_TIMEOUT;

+/* Added @ Simula */
+int sysctl_tcp_force_thin_rdb __read_mostly = TCP_FORCE_THIN_RDB;
+int sysctl_tcp_rdb_max_bundle_bytes __read_mostly = TCP_RDB_MAX_BUNDLE_BYTES;
+
 DEFINE_SNMP_STAT(struct tcp_mib, tcp_statistics) __read_mostly;

 atomic_t tcp_orphan_count = ATOMIC_INIT(0);
@@ -658,6 +662,167 @@ static inline int select_size(struct sock *sk)
        return tmp;
 }

+/* Added at Simula to support RDB */
+static int tcp_trans_merge_prev(struct sock *sk, struct sk_buff *skb, int mss_now)
+{
+        struct tcp_sock *tp = tcp_sk(sk);
+
+        /* Make sure that this isn't referenced by somebody else */
+
+        if(!skb_cloned(skb)){
+                struct sk_buff *prev_skb = skb->prev;
+                int skb_size = skb->len;
+                int old_headlen = 0;
+                int ua_data = 0;
+                int uad_head = 0;
+                int uad_frags = 0;
+                int ua_nr_frags = 0;
+                int ua_frags_diff = 0;
+
+                /* Since this technique currently does not support SACK, I
+                 * return -1 if the previous has been SACK'd. */
+                if(TCP_SKB_CB(prev_skb)->sacked & TCPCB_SACKED_ACKED){
+                        return -1;
+                }
+
+                /* Current skb is out of window. */
+                if (after(TCP_SKB_CB(skb)->end_seq, tp->snd_una+tp->snd_wnd)){
+                        return -1;
+                }
+
+                /*TODO: Optimize this part with regards to how the
+                  variables are initialized */
+
+                /*Calculates the ammount of unacked data that is available*/
+                ua_data = (TCP_SKB_CB(prev_skb)->end_seq - tp->snd_una >
+                           prev_skb->len ? prev_skb->len :
+                           TCP_SKB_CB(prev_skb)->end_seq - tp->snd_una);
+                ua_frags_diff = ua_data - prev_skb->data_len;
+                uad_frags = (ua_frags_diff > 0 ? prev_skb->data_len : ua_data);
+                uad_head = (ua_frags_diff > 0 ? ua_data - uad_frags : 0);
+
+                if(ua_data <= 0)
+                        return -1;
+
+                if(uad_frags > 0){
+                        int i = 0;
+                        int bytes_frags = 0;
+
+                        if(uad_frags == prev_skb->data_len){
+                                ua_nr_frags = skb_shinfo(prev_skb)->nr_frags;
+                        } else{
+                                for(i=skb_shinfo(prev_skb)->nr_frags - 1; i>=0; i--){
+                                        if(skb_shinfo(prev_skb)->frags[i].size
+                                           + bytes_frags == uad_frags){
+                                                ua_nr_frags += 1;
+                                                break;
+                                        }
+                                        ua_nr_frags += 1;
+                                        bytes_frags += skb_shinfo(prev_skb)->frags[i].size;
+			}
+		}
+	}
+
+	/*
+	 * Do the diffrenet checks on size and content, and return if
+	 * something will not work.
+	 *
+	 * TODO: Support copying some bytes
+	 *
+	 * 1. Larger than MSS.
+	 * 2. Enough room for the stuff stored in the linear area
+	 * 3. Enoug room for the pages
+	 * 4. If both skbs have some data stored in the linear area, and prev_skb
+	 * also has some stored in the paged area, they cannot be merged easily.
+	 * 5. If prev_skb is linear, then this one has to be it as well.
+	 */
+	if ((sysctl_tcp_rdb_max_bundle_bytes == 0 && ((skb_size + ua_data) > mss_now))
+	    || (sysctl_tcp_rdb_max_bundle_bytes > 0 && ((skb_size + ua_data) >
+							    sysctl_tcp_rdb_max_bundle_bytes))){
+		return -1;
+	}
+
+	/* We need to know tailroom, even if it is nonlinear */
+	if(uad_head > (skb->end - skb->tail)){
+		return -1;
+	}
+
+	if(skb_is_nonlinear(skb) && (uad_frags > 0)){
+		if((ua_nr_frags +
+		    skb_shinfo(skb)->nr_frags) > MAX_SKB_FRAGS){
+			return -1;
+		}
+
+		if(skb_headlen(skb) > 0){
+			return -1;
+		}
+	}
+
+	if((uad_frags > 0) && skb_headlen(skb) > 0){
+		return -1;
+	}
+
+	/* To avoid duplicate copies (and copies
+	   where parts have been acked) */
+	if(TCP_SKB_CB(skb)->seq <= (TCP_SKB_CB(prev_skb)->end_seq - ua_data)){
+		return -1;
+	}
+
+	/*SYN's are holy*/
+	if(TCP_SKB_CB(skb)->flags & TCPCB_FLAG_SYN || TCP_SKB_CB(skb)->flags & TCPCB_FLAG_FIN){
+		return -1;
+	}
+
+	/* Copy linear data */
+	if(uad_head > 0){
+
+		/* Add required space to the header. Can't use put due to linearity */
+		old_headlen = skb_headlen(skb);
```

```
+                       skb->tail += uad_head;
+                       skb->len += uad_head;
+
+                       if(skb_headlen(skb) > 0){
+                               memmove(skb->data + uad_head, skb->data, old_headlen);
+                       }
+
+                       skb_copy_to_linear_data(skb, prev_skb->data + (skb_headlen(prev_skb) - uad_head), uad_head);
+               }
+
+               /*Copy paged data*/
+               if(uad_frags > 0){
+                       int i = 0;
+                       /*Must move data backwards in the array.*/
+                       if(skb_is_nonlinear(skb)){
+                               memmove(skb_shinfo(skb)->frags + ua_nr_frags,
+                                       skb_shinfo(skb)->frags,
+                                       skb_shinfo(skb)->nr_frags*sizeof(skb_frag_t));
+                       }
+
+                       /*Copy info and update pages*/
+                       memcpy(skb_shinfo(skb)->frags,
+                              skb_shinfo(prev_skb)->frags + (skb_shinfo(prev_skb)->nr_frags - ua_nr_frags),
+                              ua_nr_frags*sizeof(skb_frag_t));
+
+                       for(i=0; i<ua_nr_frags;i++){
+                               get_page(skb_shinfo(skb)->frags[i].page);
+                       }
+
+                       skb_shinfo(skb)->nr_frags += ua_nr_frags;
+                       skb->data_len += uad_frags;
+                       skb->len += uad_frags;
+               }
+
+               TCP_SKB_CB(skb)->seq = TCP_SKB_CB(prev_skb)->end_seq - ua_data;
+
+               if(skb->ip_summed == CHECKSUM_PARTIAL)
+                       skb->csum = CHECKSUM_PARTIAL;
+               else
+                       skb->csum = skb_checksum(skb, 0, skb->len, 0);
+       }
+
+       return 1;
+}
+
 int tcp_sendmsg(struct kiocb *iocb, struct socket *sock, struct msghdr *msg,
                size_t size)
 {
@@ -825,6 +990,16 @@ new_segment:

                        from += copy;
                        copied += copy;
+
+                       /* Added at Simula to support RDB */
+                       if(((tp->thin_rdb || sysctl_tcp_force_thin_rdb)) && skb->len < mss_now){
+                               if(skb->prev != (struct sk_buff*) &(sk)->sk_write_queue
+                                  && !(TCP_SKB_CB(skb)->flags & TCPCB_FLAG_SYN)
+                                  && !(TCP_SKB_CB(skb)->flags & TCPCB_FLAG_FIN)){
+                                       tcp_trans_merge_prev(sk, skb, mss_now);
+                                }
+                        } /* End - Simula */
+
                         if ((seglen -= copy) == 0 && iovlen == 0)
                                 goto out;

@@ -1870,7 +2045,25 @@ static int do_tcp_setsockopt(struct sock *sk, int level,
                         tcp_push_pending_frames(sk);
                 }
                 break;
-
+
+         /* Added @ Simula. Support for thin streams */
+        case TCP_THIN_RDB:
+                if (val)
+                        tp->thin_rdb = 1;
+                break;
+
+         /* Added @ Simula. Support for thin streams */
+        case TCP_THIN_RM_EXPB:
+                if (val)
+                        tp->thin_rm_expb = 1;
+                break;
+
+         /* Added @ Simula. Support for thin streams */
+        case TCP_THIN_DUPACK:
+                if (val)
+                        tp->thin_dupack = 1;
+                break;
+
         case TCP_KEEPIDLE:
                 if (val < 1 || val > MAX_TCP_KEEPIDLE)
                         err = -EINVAL;
diff --git a/net/ipv4/tcp_input.c b/net/ipv4/tcp_input.c
index f893e90..f42ef14 100644
--- a/net/ipv4/tcp_input.c
+++ b/net/ipv4/tcp_input.c
@@ -89,6 +89,9 @@ int sysctl_tcp_frto __read_mostly;
 int sysctl_tcp_frto_response __read_mostly;
 int sysctl_tcp_nometrics_save __read_mostly;

+/* Added @ Simula */
+int sysctl_tcp_force_thin_dupack __read_mostly = TCP_FORCE_THIN_DUPACK;
+
 int sysctl_tcp_moderate_rcvbuf __read_mostly = 1;
 int sysctl_tcp_abc __read_mostly;

@@ -1704,6 +1707,12 @@ static int tcp_time_to_recover(struct sock *sk)
                  */
                 return 1;
         }
+
+        /*Added at Simula to modify fast retransmit */
+        if ((tp->thin_dupack || sysctl_tcp_force_thin_dupack) &&
+            tcp_fackets_out(tp) > 1 && tcp_stream_is_thin(tp)){
+          return 1;
+        }

         return 0;
 }
@@ -2437,30 +2446,127 @@ static int tcp_clean_rtx_queue(struct sock *sk, __s32 *seq_rtt_p)
 {
         struct tcp_sock *tp = tcp_sk(sk);
         const struct inet_connection_sock *icsk = inet_csk(sk);
-        struct sk_buff *skb;
+        struct sk_buff *skb = tcp_write_queue_head(sk);
+        struct sk_buff *next_skb;
+
         __u32 now = tcp_time_stamp;
         int acked = 0;
         int prior_packets = tp->packets_out;
+
+        /*Added at Simula for RDB support*/
+        __u8 done = 0;
+        int remove = 0;
+        int remove_head = 0;
+        int remove_frags = 0;
+        int no_frags;
+        int data_frags;
+        int i;
+
         __s32 seq_rtt = -1;
         ktime_t last_ackt = net_invalid_timestamp();
-
-        while ((skb = tcp_write_queue_head(sk)) &&
-               skb != tcp_send_head(sk)) {
+
+        while (skb != NULL
+              && ((!(tp->thin_rdb || sysctl_tcp_force_thin_rdb)
+                   && skb != tcp_send_head(sk)
+                   && skb != (struct sk_buff *)&sk->sk_write_queue)
+                  || ((tp->thin_rdb || sysctl_tcp_force_thin_rdb)
+                      && skb != (struct sk_buff *)&sk->sk_write_queue))){
                 struct tcp_skb_cb *scb = TCP_SKB_CB(skb);
                 __u8 sacked = scb->sacked;
-
+
+
+                if(skb == NULL){
+                        break;
+                }
+
+                if(skb == tcp_send_head(sk)){
+                        break;
+                }
+
+                if(skb == (struct sk_buff *)&sk->sk_write_queue){
+                        break;
+                }
+
                 /* If our packet is before the ack sequence we can
                  * discard it as it's confirmed to have arrived at
                  * the other end.
                  */
                 if (after(scb->end_seq, tp->snd_una)) {
-                        if (tcp_skb_pcount(skb) > 1 &&
-                            after(tp->snd_una, scb->seq))
-                                acked |= tcp_tso_acked(sk, skb,
-                                                       now, &seq_rtt);
-                       break;
+                       if (tcp_skb_pcount(skb) > 1 && after(tp->snd_una, scb->seq))
+                               acked |= tcp_tso_acked(sk, skb, now, &seq_rtt);
+
+                       done = 1;
+
+                       /* Added at Simula for RDB support*/
+                       if ((tp->thin_rdb || sysctl_tcp_force_thin_rdb) && after(tp->snd_una, scb->seq)) {
+                               if (!skb_cloned(skb) && !(scb->flags & TCPCB_FLAG_SYN)){
+                                       remove = tp->snd_una - scb->seq;
+                                       remove_head = (remove > skb_headlen(skb) ?
+                                                      skb_headlen(skb) : remove);
+                                       remove_frags = (remove > skb_headlen(skb) ?
+                                                       remove - remove_head : 0);
+
+                                       /* Has linear data */
+                                       if(skb_headlen(skb) > 0 && remove_head > 0){
+                                               memmove(skb->data,
+                                                       skb->data + remove_head,
+                                                       skb_headlen(skb) - remove_head);
+
+                                               skb->tail -= remove_head;
+                                       }
+
+                                       if(skb_is_nonlinear(skb) && remove_frags > 0){
+                                               no_frags = 0;
+                                               data_frags = 0;
+
+                                               /*Remove unecessary pages*/
+                                               for(i=0; i<skb_shinfo(skb)->nr_frags; i++){
+                                                       if(data_frags + skb_shinfo(skb)->frags[i].size
+                                                          == remove_frags){
+                                                               put_page(skb_shinfo(skb)->frags[i].page);
+                                                               no_frags += 1;
+                                                               break;
+                                                       }
+                                                       put_page(skb_shinfo(skb)->frags[i].page);
+                                                       no_frags += 1;
+                                                       data_frags += skb_shinfo(skb)->frags[i].size;
+                                               }
+
+                                               if(skb_shinfo(skb)->nr_frags > no_frags)
+                                                       memmove(skb_shinfo(skb)->frags,
+                                                               skb_shinfo(skb)->frags + no_frags,
+                                                               (skb_shinfo(skb)->nr_frags
+                                                                - no_frags)*sizeof(skb_frag_t));
+
+                                               skb->data_len -= remove_frags;
+                                               skb_shinfo(skb)->nr_frags -= no_frags;
+
+                                       }
+
+                                       scb->seq += remove;
+                                       skb->len -= remove;
+
+                                       if(skb->ip_summed == CHECKSUM_PARTIAL)
+                                               skb->csum = CHECKSUM_PARTIAL;
+                                       else
+                                               skb->csum = skb_checksum(skb, 0, skb->len, 0);
+
+                                }
+
+                                /*Only move forward if data could be removed from this packet*/
+                                done = 2;
+
+                        }
+
+                        if(done == 1 || tcp_skb_is_last(sk,skb)){
+                                break;
+                        } else if(done == 2){
+                                skb = skb->next;
+                                done = 1;
+                                continue;
+                        }
+
                 }
-
+
                 /* Initial outgoing SYN's get put onto the write_queue
                  * just like anything else we transmit.  It is not
                  * true data, and if we misinform our callers that
@@ -2474,14 +2580,14 @@ static int tcp_clean_rtx_queue(struct sock *sk, __s32 *seq_rtt_p)
                         acked |= FLAG_SYN_ACKED;
                         tp->retrans_stamp = 0;
                 }
-
+
                 /* MTU probing checks */
                 if (icsk->icsk_mtup.probe_size) {
                         if (!after(tp->mtu_probe.probe_seq_end, TCP_SKB_CB(skb)->end_seq)) {
                                 tcp_mtup_probe_success(sk, skb);
                         }
                 }
-
+
                 if (sacked) {
                         if (sacked & TCPCB_RETRANS) {
                                 if (sacked & TCPCB_SACKED_RETRANS)
@@ -2505,24 +2611,32 @@ static int tcp_clean_rtx_queue(struct sock *sk, __s32 *seq_rtt_p)
                         seq_rtt = now - scb->when;
                         last_ackt = skb->tstamp;
                 }
+
+                if ((tp->thin_rdb || sysctl_tcp_force_thin_rdb) && skb == tcp_send_head(sk)) {
+                        tcp_advance_send_head(sk, skb);
+                }
+
                 tcp_dec_pcount_approx(&tp->fackets_out, skb);
                 tcp_packets_out_dec(tp, skb);
+                next_skb = skb->next;
                 tcp_unlink_write_queue(skb, sk);
                 sk_stream_free_skb(sk, skb);
                 clear_all_retrans_hints(tp);
+                /* Added at Simula to support RDB */
+                skb = next_skb;
         }
-
+
         if (acked&FLAG_ACKED) {
                 u32 pkts_acked = prior_packets - tp->packets_out;
                 const struct tcp_congestion_ops *ca_ops
                         = inet_csk(sk)->icsk_ca_ops;
-
+
                 tcp_ack_update_rtt(sk, acked, seq_rtt);
                 tcp_ack_packets_out(sk);
-
+
                 if (ca_ops->pkts_acked) {
                         s32 rtt_us = -1;
-
+
                         /* Is the ACK triggering packet unambiguous? */
                         if (!(acked & FLAG_RETRANS_DATA_ACKED)) {
                                 /* High resolution needed and available? */
diff --git a/net/ipv4/tcp_output.c b/net/ipv4/tcp_output.c
index 666d8a5..daa580d 100644
--- a/net/ipv4/tcp_output.c
+++ b/net/ipv4/tcp_output.c
@@ -1653,7 +1653,7 @@ static void tcp_retrans_try_collapse(struct sock *sk, struct sk_buff *skb, int m

                 BUG_ON(tcp_skb_pcount(skb) != 1 ||
                        tcp_skb_pcount(next_skb) != 1);
-
+
                 /* changing transmit queue under us so clear hints */
                 clear_all_retrans_hints(tp);

@@ -1702,6 +1702,166 @@ static void tcp_retrans_try_collapse(struct sock *sk, struct sk_buff *skb, int m
         }
 }

+/* Added at Simula. Variation of the regular collapse,
+   adapted to support RDB  */
+static void tcp_retrans_merge_redundant(struct sock *sk,
+                                        struct sk_buff *skb,
+                                        int mss_now)
+{
+        struct tcp_sock *tp = tcp_sk(sk);
+        struct sk_buff *next_skb = skb->next;
+        int skb_size = skb->len;
+        int new_data = 0;
+        int new_data_head = 0;
+        int new_data_frags = 0;
+        int new_frags = 0;
+        int old_headlen = 0;
+
+        int i;
+        int data_frags = 0;
+
+        /* Loop through as many packets as possible
+         * (will create a lot of redundant data, but WHATEVER).
+         * The only packet this MIGHT be critical for is
+         * if this packet is the last in the retrans-queue.
+         *
+         * Make sure that the first skb isnt already in
+         * use by somebody else. */
```

```
+
+        if (!skb_cloned(skb)) {
+                /* Iterate through the retransmit queue */
+                for (; (next_skb != (sk)->sk_send_head) &&
+                             (next_skb != (struct sk_buff *) &(sk)->sk_write_queue);
+                     next_skb = next_skb->next) {
+
+                        /* Reset variables */
+                        new_frags = 0;
+                        data_frags = 0;
+                        new_data = TCP_SKB_CB(next_skb)->end_seq - TCP_SKB_CB(skb)->end_seq;
+
+                        /* New data will be stored at skb->start_add + some_offset,
+                           in other words the last N bytes */
+                        new_data_frags = (new_data > next_skb->data_len ?
+                                          next_skb->data_len : new_data);
+                        new_data_head = (new_data > next_skb->data_len ?
+                                         new_data - skb->data_len : 0);
+
+                        /*
+                         * 1. Contains the same data
+                         * 2. Size
+                         * 3. Sack
+                         * 4. Window
+                         * 5. Cannot merge with a later packet that has linear data
+                         * 6. The new number of frags will exceed the limit
+                         * 7. Enough tailroom
+                         */
+
+                        if(new_data <= 0){
+                                return;
+                        }
+
+                        if ((sysctl_tcp_rdb_max_bundle_bytes == 0 && ((skb_size + new_data) > mss_now))
+                            || (sysctl_tcp_rdb_max_bundle_bytes > 0 && ((skb_size + new_data) >
+                                                                        sysctl_tcp_rdb_max_bundle_bytes))){
+                                return;
+                        }
+
+                        if(TCP_SKB_CB(next_skb)->flags & TCPCB_FLAG_FIN){
+                                return;
+                        }
+
+                        if((TCP_SKB_CB(skb)->sacked & TCPCB_SACKED_ACKED) ||
+                           (TCP_SKB_CB(next_skb)->sacked & TCPCB_SACKED_ACKED)){
+                                return;
+                        }
+
+                        if(after(TCP_SKB_CB(skb)->end_seq + new_data, tp->snd_una + tp->snd_wnd)){
+                                return;
+                        }
+
+                        if(skb_shinfo(skb)->frag_list || skb_shinfo(skb)->frag_list){
+                                return;
+                        }
+
+                        /* Calculate number of new fragments. Any new data will be
+                           stored in the back. */
+                        if(skb_is_nonlinear(next_skb)){
```

```
+            i = (skb_shinfo(next_skb)->nr_frags == 0 ?
+                0 : skb_shinfo(next_skb)->nr_frags - 1);
+            for( ; i>=0;i--){
+                if(data_frags + skb_shinfo(next_skb)->frags[i].size ==
+                  new_data_frags){
+                    new_frags += 1;
+                    break;
+                }
+
+                data_frags += skb_shinfo(next_skb)->frags[i].size;
+                new_frags += 1;
+            }
+        }
+
+        /* If dealing with a fragmented skb, only merge
+           with an skb that ONLY contain frags */
+        if(skb_is_nonlinear(skb)){
+
+            /*Due to the way packets are processed, no later data*/
+            if(skb_headlen(next_skb) && new_data_head > 0){
+                return;
+            }
+
+            if(skb_is_nonlinear(next_skb) && (new_data_frags > 0) &&
+              ((skb_shinfo(skb)->nr_frags + new_frags) > MAX_SKB_FRAGS)){
+                return;
+            }
+
+        } else {
+            if(skb_headlen(next_skb) && (new_data_head > (skb->end - skb->tail))){
+                return;
+            }
+        }
+
+        /*Copy linear data. This will only occur if both are linear,
+          or only A is linear*/
+        if(skb_headlen(next_skb) && (new_data_head > 0)){
+            old_headlen = skb_headlen(skb);
+            skb->tail += new_data_head;
+            skb->len += new_data_head;
+
+            /* The new data starts in the linear area,
+              and the correct offset will then be given by
+              removing new_data ammount of bytes from length. */
+            skb_copy_to_linear_data_offset(skb, old_headlen, next_skb->tail -
+                                           new_data_head, new_data_head);
+        }
+
+        if(skb_is_nonlinear(next_skb) && (new_data_frags > 0)){
+            memcpy(skb_shinfo(skb)->frags + skb_shinfo(skb)->nr_frags,
+                  skb_shinfo(next_skb)->frags +
+                  (skb_shinfo(next_skb)->nr_frags - new_frags),
+                  new_frags*sizeof(skb_frag_t));
+
+            for(i=skb_shinfo(skb)->nr_frags;
+                i < skb_shinfo(skb)->nr_frags + new_frags; i++)
+                get_page(skb_shinfo(skb)->frags[i].page);
+
+            skb_shinfo(skb)->nr_frags += new_frags;
+                                       skb->data_len += new_data_frags;
+                                       skb->len += new_data_frags;
+                               }
+
+                               TCP_SKB_CB(skb)->end_seq += new_data;
+
+                               if(skb->ip_summed == CHECKSUM_PARTIAL)
+                                       skb->csum = CHECKSUM_PARTIAL;
+                               else
+                                       skb->csum = skb_checksum(skb, 0, skb->len, 0);
+
+                               skb_size = skb->len;
+                       }
+
+               }
+}
+
 /* Do a simple retransmit without using the backoff mechanisms in
  * tcp_timer. This is used for path mtu discovery.
  * The socket is already locked here.
@@ -1756,6 +1916,8 @@ void tcp_simple_retransmit(struct sock *sk)
 /* This retransmits one SKB.  Policy decisions and retransmit queue
  * state updates are done by the caller.  Returns non-zero if an
  * error occurred which prevented the send.
+ * Modified at Simula to support thin stream optimizations
+ * TODO: Update to use new helpers (like tcp_write_queue_next())
  */
 int tcp_retransmit_skb(struct sock *sk, struct sk_buff *skb)
 {
@@ -1802,10 +1964,21 @@ int tcp_retransmit_skb(struct sock *sk, struct sk_buff *skb)
                    (skb->len < (cur_mss >> 1)) &&
                    (tcp_write_queue_next(sk, skb) != tcp_send_head(sk)) &&
                    (!tcp_skb_is_last(sk, skb)) &&
-                   (skb_shinfo(skb)->nr_frags == 0 && skb_shinfo(tcp_write_queue_next(sk, skb))->nr_frags == 0) &&
-                   (tcp_skb_pcount(skb) == 1 && tcp_skb_pcount(tcp_write_queue_next(sk, skb)) == 1) &&
-                   (sysctl_tcp_retrans_collapse != 0))
+                   (skb_shinfo(skb)->nr_frags == 0
+                    && skb_shinfo(tcp_write_queue_next(sk, skb))->nr_frags == 0)
+                   && (tcp_skb_pcount(skb) == 1
+                       && tcp_skb_pcount(tcp_write_queue_next(sk, skb)) == 1)
+                   && (sysctl_tcp_retrans_collapse != 0)
+                   && !((tp->thin_rdb || sysctl_tcp_force_thin_rdb))) {
                        tcp_retrans_try_collapse(sk, skb, cur_mss);
+                       } else if ((tp->thin_rdb || sysctl_tcp_force_thin_rdb)) {
+                       if (!(TCP_SKB_CB(skb)->flags & TCPCB_FLAG_SYN) &&
+                           !(TCP_SKB_CB(skb)->flags & TCPCB_FLAG_FIN) &&
+                           (skb->next != tcp_send_head(sk)) &&
+                           (skb->next != (struct sk_buff *) &sk->sk_write_queue)) {
+                               tcp_retrans_merge_redundant(sk, skb, cur_mss);
+                       }
+               }

                if (inet_csk(sk)->icsk_af_ops->rebuild_header(sk))
                        return -EHOSTUNREACH; /* Routing failure or similar. */
diff --git a/net/ipv4/tcp_timer.c b/net/ipv4/tcp_timer.c
index e9b151b..ad8de35 100644
--- a/net/ipv4/tcp_timer.c
+++ b/net/ipv4/tcp_timer.c
@@ -32,6 +32,9 @@ int sysctl_tcp_retries1 __read_mostly = TCP_RETR1;
```

```
 int sysctl_tcp_retries2 __read_mostly = TCP_RETR2;
 int sysctl_tcp_orphan_retries __read_mostly;

+/* Added @ Simula */
+int sysctl_tcp_force_thin_rm_expb __read_mostly = TCP_FORCE_THIN_RM_EXPB;
+
 static void tcp_write_timer(unsigned long);
 static void tcp_delack_timer(unsigned long);
 static void tcp_keepalive_timer (unsigned long data);
@@ -368,13 +371,28 @@ static void tcp_retransmit_timer(struct sock *sk)
          */
        icsk->icsk_backoff++;
        icsk->icsk_retransmits++;
-
+
 out_reset_timer:
-       icsk->icsk_rto = min(icsk->icsk_rto << 1, TCP_RTO_MAX);
+       /* Added @ Simula removal of exponential backoff for thin streams
+          We only want to apply this for an established stream */
+       if ((tp->thin_rm_expb || sysctl_tcp_force_thin_rm_expb)
+           && tcp_stream_is_thin(tp) && sk->sk_state == TCP_ESTABLISHED) {
+               /* Since 'icsk_backoff' is used to reset timer, set to 0
+                * Recalculate 'icsk_rto' as this might be increased if stream oscillates
+                * between thin and thick, thus the old value might already be too high
+                * compared to the value set by 'tcp_set_rto' in tcp_input.c which resets
+                * the rto without backoff. */
+               icsk->icsk_backoff = 0;
+               icsk->icsk_rto = min(((tp->srtt >> 3) + tp->rttvar), TCP_RTO_MAX);
+       } else {
+               /* Use normal backoff */
+               icsk->icsk_rto = min(icsk->icsk_rto << 1, TCP_RTO_MAX);
+       }
+       /* End Simula*/
        inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS, icsk->icsk_rto, TCP_RTO_MAX);
        if (icsk->icsk_retransmits > sysctl_tcp_retries1)
                __sk_dst_reset(sk);
-
+
 out:;
 }

--
1.5.6.3
```

Appendix F

SCTP-patch for Linux 2.6.16 kernel

```
diff -Naur linux-2.6.16.13/include/net/sctp/structs.h linux-2.6.16.13-modSCTP/include/net/sctp/structs.h
--- linux-2.6.16.13/include/net/sctp/structs.h  2006-05-02 23:38:44.000000000 +0200
+++ linux-2.6.16.13-modSCTP/include/net/sctp/structs.h  2007-07-16 16:15:03.000000000 +0200
@@ -1,6 +1,6 @@
 /* SCTP kernel reference Implementation
  * (C) Copyright IBM Corp. 2001, 2004
- * Copyright (c) 1999-2000 Cisco, Inc.
+ * Copyright (c) 1999-2000, 2007 Cisco, Inc.
  * Copyright (c) 1999-2001 Motorola, Inc.
  * Copyright (c) 2001 Intel Corp.
  *
@@ -75,6 +75,7 @@
 };

 /* Forward declarations for data structures. */
+struct pkt_stat; // Added by Simula
 struct sctp_globals;
 struct sctp_endpoint;
 struct sctp_association;
@@ -93,6 +94,28 @@
 #include <net/sctp/ulpevent.h>
 #include <net/sctp/ulpqueue.h>

+
+/* Structure for holding packet stats, added by Simula */
+
+struct pkt_stat{
+
+  int is_gap_acked; /* Is this packet received? (highest TSN in packet is gap acked) */
+  int marked_lost; /* Is this packet lost? */
+
+  /* Number of times the data chunk with highest
+     TSN in packet is indicated lost by SACK gap
+     3 loss indications (as in fast retransmit) marks
+     packet as lost
+  */
+  int indicated_lost;
+
+  unsigned long timestamp; /* Timestamp of when this packet is sent */
+
+  __u32 highestTSN; /* */
+  struct pkt_stat *next, *prev;
+};
+
+
 /* Structures useful for managing bind/connect. */

 struct sctp_bind_bucket {
@@ -213,8 +236,32 @@

        /* Flag to indicate if PR-SCTP is enabled. */
        int prsctp_enable;
+
+
+  /* Added by Simula:
+      Turn on/off thin stream mechanisms
+  */
+
+  int thin_fr; // Fast retransmit after 1 SACK
+  int thin_minrto; //Reduce RTO min to 200 ms
+  int thin_bundling_outstanding; // Bundle outstanding data chunks
+  int thin_bundling_fr; // Bundle in fast retransmits
+  int thin_expbackoff; // Avoid exp backoff in thin streams
+  int thin_restart_timer; //Restart timer modification
+  int thin_debug_tag_payload; //Trigger tagging of payload to be able to accurately determine retransmission reason
+
 } sctp_globals;

+/* Added by Simula: Thin stream proc variables*/
+
+#define sctp_thin_fr                      (sctp_globals.thin_fr)
+#define sctp_thin_minrto                  (sctp_globals.thin_minrto)
+#define sctp_thin_bundling_outstanding    (sctp_globals.thin_bundling_outstanding)
+#define sctp_thin_bundling_fr             (sctp_globals.thin_bundling_fr)
+#define sctp_thin_expbackoff              (sctp_globals.thin_expbackoff)
+#define sctp_thin_restart_timer           (sctp_globals.thin_restart_timer)
+#define sctp_thin_debug_tag_payload       (sctp_globals.thin_debug_tag_payload)
+
 #define sctp_rto_initial                 (sctp_globals.rto_initial)
 #define sctp_rto_min                     (sctp_globals.rto_min)
 #define sctp_rto_max                     (sctp_globals.rto_max)
@@ -732,7 +779,7 @@
 /* This structure holds lists of chunks as we are assembling for
  * transmission.
  */
-struct sctp_packet {
+struct sctp_packet{
        /* These are the SCTP header values (host order) for the packet. */
        __u16 source_port;
        __u16 destination_port;
@@ -827,6 +874,12 @@
         */
        __u32 rtt;              /* This is the most recent RTT.  */

+        /* Added by Simula to find the min RTT used in packet loss
+           rate calculation.
+       */
+        __u32 min_rtt;
+
+
        /* RTO          : The current retransmission timeout value.  */
        unsigned long rto;

@@ -1335,6 +1388,20 @@
        /* These are those association elements needed in the cookie.  */
        struct sctp_cookie c;

+        /* added by Simula to count packets in flight and packet loss rate */
+        int packets_in_flight;
+        int packets_left_network;
+        struct pkt_stat *pkt_stat_head, *pkt_stat_tail;
+
+        /* Added by Simula: Variables for holding lost packets and total xmited
+           packets used for packet loss rate calculation and adjustment of
+           thin stream threshold for this association.
+       */
+
+        int lost_packets;
+        int total_xmt_packets;
+        int thin_stream_threshold;
+
        /* This is all information about our peer.  */
        struct {
                /* rwnd
 diff -Naur linux-2.6.16.13/net/sctp/associola.c linux-2.6.16.13-modSCTP/net/sctp/associola.c
--- linux-2.6.16.13/net/sctp/associola.c        2006-05-02 23:38:44.000000000 +0200
+++ linux-2.6.16.13-modSCTP/net/sctp/associola.c        2007-07-16 15:57:59.000000000 +0200
@@ -1,6 +1,6 @@
 /* SCTP kernel reference Implementation
  * (C) Copyright IBM Corp. 2001, 2004
- * Copyright (c) 1999-2000 Cisco, Inc.
+ * Copyright (c) 1999-2000, 2006 Cisco, Inc.
  * Copyright (c) 1999-2001 Motorola, Inc.
  * Copyright (c) 2001 Intel Corp.
  * Copyright (c) 2001 La Monte H.P. Yarroll
@@ -112,6 +112,19 @@
                                                * 1000;
        asoc->frag_point = 0;

+       /* added by Simula to initialize packets in flight and packet loss datastructure */
+
+       asoc->packets_in_flight = 0;
+       asoc->packets_left_network = 0;
+       asoc->pkt_stat_head = NULL;
+       asoc->pkt_stat_tail = NULL;
+
+       asoc->lost_packets = 0;
+       asoc->total_xmt_packets = 0;
+
+       printk(KERN_ALERT "[SCTP]␣Init␣association\n");
+
+
        /* Set the association max_retrans and RTO values from the
         * socket values.
         */
 diff -Naur linux-2.6.16.13/net/sctp/output.c linux-2.6.16.13-modSCTP/net/sctp/output.c
--- linux-2.6.16.13/net/sctp/output.c    2006-05-02 23:38:44.000000000 +0200
+++ linux-2.6.16.13-modSCTP/net/sctp/output.c    2007-07-16 15:57:59.000000000 +0200
@@ -1,6 +1,6 @@
```

```
 /* SCTP kernel reference Implementation
  * (C) Copyright IBM Corp. 2001, 2004
- * Copyright (c) 1999-2000 Cisco, Inc.
+ * Copyright (c) 1999-2000, 2006 Cisco, Inc.
  * Copyright (c) 1999-2001 Motorola, Inc.
  *
  * This file is part of the SCTP kernel reference Implementation
@@ -65,6 +65,13 @@
 static sctp_xmit_t sctp_packet_append_data(struct sctp_packet *packet,
                                            struct sctp_chunk *chunk);

+/* Routines added by Simula */
+
+void add_pkt_in_flight(struct sctp_association *a, __u32 tsn);
+int calculate_packet_loss_rate(struct sctp_association *a);
+int calculate_thin_stream_threshold(int loss_rate);
+
+
 /* Config a packet.
  * This appears to be a followup set of initializations.
  */
@@ -303,6 +310,12 @@
        int padding;            /* How much padding do we need?  */
        __u8 has_data = 0;
        struct dst_entry *dst;
+
+
+       /* Added by Simula to keep track of highest TSN in packet */
+       __u32 pktHighestTSN = 0;
+       __u32 currentTSN;
+

        SCTP_DEBUG_PRINTK("%s:␣packet:%p\n", __FUNCTION__, packet);

@@ -396,6 +409,15 @@

                        chunk->sent_at = jiffies;
                        has_data = 1;
+
+                       currentTSN = ntohl(chunk->subh.data_hdr->tsn);
+
+                       /* Added by Simula to find highest TSN in packet containing
+                          data chunks: */
+                       if(currentTSN > pktHighestTSN){
+                         pktHighestTSN = currentTSN;
+                       }
+
                }

                padding = WORD_ROUND(chunk->skb->len) - chunk->skb->len;
@@ -475,6 +497,17 @@
                        if (!mod_timer(timer, jiffies + timeout))
                                sctp_association_hold(asoc);
                }
+
+               /* added by Simula to add packet to packets in flight list */
+               add_pkt_in_flight(asoc, pktHighestTSN);
+               asoc->total_xmt_packets++;
+
+               /* added by Simula:
+                  Raises thin stream threshold if loss rate increases.*/
+               if(asoc->total_xmt_packets%30 == 0)
+                 asoc->thin_stream_threshold = calculate_thin_stream_threshold( calculate_packet_loss_rate(asoc) / 100 );
+
+
        }

        dst = tp->dst;
@@ -653,3 +686,75 @@
 finish:
        return retval;
 }
+
+
+/* added by Simula:
+   This routine adds a new packet with its highest TSN to the packets_in_flight
+   list pointed to by pkt_in_flight_head. Finally, packets_in_flight is
+   incremented by one. Finally, loss reports and a sendtime timestamp are set.
+
+   The first parameter is a pointer to the current
+   sctp_association holding the 'packets in flight' datastructure.
+   The second parameter is the highest TSN in the packet.
+
+*/
+void add_pkt_in_flight(struct sctp_association *a, __u32 tsn){
+
+  struct pkt_stat *cur;
+
+  if(a->pkt_stat_head == NULL){
+    a->pkt_stat_head = (struct pkt_stat*) kmalloc(sizeof(struct pkt_stat), GFP_KERNEL);
+    a->pkt_stat_head->next = NULL;
+    a->pkt_stat_head->prev = NULL;
+    a->pkt_stat_tail = a->pkt_stat_head;
+
+    cur = a->pkt_stat_head;
+  }
+  else{
+    a->pkt_stat_tail->next = (struct pkt_stat*) kmalloc(sizeof(struct pkt_stat), GFP_KERNEL);
+    a->pkt_stat_tail->next->prev = a->pkt_stat_tail;
+    a->pkt_stat_tail = a->pkt_stat_tail->next;
+    a->pkt_stat_tail->next = NULL;
+
+    cur = a->pkt_stat_tail;
+  }
+
+  cur->highestTSN = tsn;
+  cur->timestamp = jiffies;
+  cur->is_gap_acked = 0;
+  cur->marked_lost = 0;
+  cur->indicated_lost = 0;
+
+  a->packets_in_flight++;
+}
+
+
+/* This routine calculates the packet loss rate by integer
+   division. To get the actual loss rate in percent,
+   divide the loss rate by 100.
```

```
+*/
+
+int calculate_packet_loss_rate(struct sctp_association *a){
+
+  int lpackets, total_xmt;
+
+  lpackets = a->lost_packets * 10000;
+  total_xmt = a->total_xmt_packets;
+
+  /* printk(KERN_ALERT "lpackets: %d, total_xmt: %d, rate: %d\n",
+      a->lost_packets, total_xmt, (lpackets / total_xmt)); */
+
+  return lpackets / total_xmt;
+}
+
+
+int calculate_thin_stream_threshold(int loss_rate){
+  int piflim;
+
+  piflim = (4 * (1000000 / (10000 - ( (loss_rate * 10000) / 100 )))/100);
+
+  printk(KERN_ALERT "loss_rate: %i, piflim: %i\n",
+          loss_rate, piflim);
+
+  return piflim;
+}
diff -Naur linux-2.6.16.13/net/sctp/outqueue.c linux-2.6.16.13-modSCTP/net/sctp/outqueue.c
--- linux-2.6.16.13/net/sctp/outqueue.c 2006-05-02 23:38:44.000000000 +0200
+++ linux-2.6.16.13-modSCTP/net/sctp/outqueue.c 2007-07-16 16:42:48.000000000 +0200
@@ -1,6 +1,6 @@
 /* SCTP kernel reference Implementation
  * (C) Copyright IBM Corp. 2001, 2004
- * Copyright (c) 1999-2000 Cisco, Inc.
+ * Copyright (c) 1999-2000, 2006, 2007 Cisco, Inc.
  * Copyright (c) 1999-2001 Motorola, Inc.
  * Copyright (c) 2001-2003 Intel Corp.
  *
@@ -56,6 +56,25 @@
 #include <net/sctp/sm.h>

 /* Declare internal functions here. */
+
+/* Packet loss rate added by Simula */
+void update_lost_packets_by_timeout(struct sctp_transport *t);
+void update_lost_packets_by_sackgap(struct sctp_association *a, struct sctp_sackhdr *sack);
+void remove_pkts_in_flight(struct sctp_association *a, __u32 sack_cum_tsn);
+void loss_indication_update(struct sctp_association *a, __u32 gap_tsn);
+void mark_gap_acked(struct sctp_association *a, __u32 gap_acked_tsn);
+//int is_spurious(struct sctp_transport *t, struct pkt_stat *c);
+
+/* 3 thin stream routines added by Simula: */
+void remove_pkts_in_flight(struct sctp_association *a, __u32 sack_cum_tsn);
+int check_stream_before_add(struct sctp_transport *t,
+                                   struct sctp_chunk *chunk, __u8 fast_retransmit);
+void bundle_outstanding_chunks(struct sctp_packet *packet,
+                                 struct sctp_transport *transport);
+
+
+
+
 static int sctp_acked(struct sctp_sackhdr *sack, __u32 tsn);
 static void sctp_check_transmitted(struct sctp_outq *q,
                                    struct list_head *transmitted_queue,
@@ -379,6 +398,38 @@
                 list_add_tail(new, head);
 }
 
+
+/* added by Simula. When the stream is thin, this routine makes it possible for the
+   sender to also bundle outstanding chunks with chunks marked for fast retransmit
+   in addition to chunks that will get retransmitted due to a retransmission timeout.
+   If the stream is thick, outstanding chunks will only get bundled with chunks
+   that will get retransmitted due to a retransmission timeout.
+*/
+
+int check_stream_before_add(struct sctp_transport *t,
+                            struct sctp_chunk *chunk,
+                            __u8 fast_retransmit){
+  /* Check proc variable */
+
+  if(sctp_thin_bundling_fr){
+
+    /* Less than thin_stream_threshold packets in flight */
+    if(t->asoc->packets_in_flight < t->asoc->thin_stream_threshold){
+      return ((fast_retransmit && chunk->fast_retransmit) ||
+             !chunk->tsn_gap_acked);
+    }
+    else{
+      return ((fast_retransmit && chunk->fast_retransmit) ||
+             (!fast_retransmit && !chunk->tsn_gap_acked));
+    }
+  }
+  else{
+    return ((fast_retransmit && chunk->fast_retransmit) ||
+           (!fast_retransmit && !chunk->tsn_gap_acked));
+  }
+}
+
+
 /* Mark all the eligible packets on a transport for retransmission.  */
 void sctp_retransmit_mark(struct sctp_outq *q,
                           struct sctp_transport *transport,
@@ -387,65 +438,156 @@
         struct list_head *lchunk, *ltemp;
         struct sctp_chunk *chunk;
 
-        /* Walk through the specified transmitted queue.  */
-        list_for_each_safe(lchunk, ltemp, &transport->transmitted) {
-                chunk = list_entry(lchunk, struct sctp_chunk,
-                                   transmitted_list);
-
-                /* If the chunk is abandoned, move it to abandoned list. */
-                if (sctp_chunk_abandoned(chunk)) {
-                        list_del_init(lchunk);
-                        sctp_insert_list(&q->abandoned, lchunk);
-                        continue;
-                }
-
-               /* If we are doing retransmission due to a fast retransmit,
-                * only the chunk's that are marked for fast retransmit
-                * should be added to the retransmit queue.  If we are doing
-                * retransmission due to a timeout or pmtu discovery, only the
-                * chunks that are not yet acked should be added to the
-                * retransmit queue.
-                */
-               if ((fast_retransmit && (chunk->fast_retransmit > 0)) ||
-                  (!fast_retransmit && !chunk->tsn_gap_acked)) {
-                       /* RFC 2960 6.2.1 Processing a Received SACK
-                        *
-                        * C) Any time a DATA chunk is marked for
-                        * retransmission (via either T3-rtx timer expiration
-                        * (Section 6.3.3) or via fast retransmit
-                        * (Section 7.2.4)), add the data size of those
-                        * chunks to the rwnd.
-                        */
-                       q->asoc->peer.rwnd += sctp_data_size(chunk);
-                       q->outstanding_bytes -= sctp_data_size(chunk);
-                       transport->flight_size -= sctp_data_size(chunk);
-
-                       /* sctpimpguide-05 Section 2.8.2
-                        * M5) If a T3-rtx timer expires, the
-                        * 'TSN.Missing.Report' of all affected TSNs is set
-                        * to 0.
-                        */
-                       chunk->tsn_missing_report = 0;

-                       /* If a chunk that is being used for RTT measurement
-                        * has to be retransmitted, we cannot use this chunk
-                        * anymore for RTT measurements. Reset rto_pending so
-                        * that a new RTT measurement is started when a new
-                        * data chunk is sent.
-                        */
-                       if (chunk->rtt_in_progress) {
-                               chunk->rtt_in_progress = 0;
-                               transport->rto_pending = 0;
-                       }
+       /* added by jonped */
+       struct list_head *dchunk, *ttemp;
+       struct sctp_chunk *tchunk;

-                       /* Move the chunk to the retransmit queue. The chunks
-                        * on the retransmit queue are always kept in order.
-                        */
-                       list_del_init(lchunk);
-                       sctp_insert_list(&q->retransmit, lchunk);
-               }
+       __u32 lowestTSN = 0;
+       __u32 currentTSN = 0;
+
+       char *data_chunk_payload;
+       int chunk_offset;
+
+       /*
+          added by Simula:
+
+          Try to find the lowest TSN on the transmitted queue. The transmitted queue COULD
+          be unordered as retransmitted data chunks is are put back on the transmitted queue,
+      but it is the data chunk with the lowest TSN that caused the retransmission
+      timeout.
+   */
+
+   if(!fast_retransmit){
+     list_for_each_safe(dchunk, ttemp, &transport->transmitted) {
+
+       tchunk = list_entry(dchunk, struct sctp_chunk,
+                           transmitted_list);
+
+       if(!tchunk->tsn_gap_acked){
+
+         currentTSN = ntohl(tchunk->subh.data_hdr->tsn);
+
+         if(!lowestTSN){
+           lowestTSN = currentTSN;
+         }
+         else{
+           if(currentTSN < lowestTSN){
+             lowestTSN = currentTSN;
+           }
+         }
+       }
+     }
    }
−
+
+   /* Walk through the specified transmitted queue.  */
+   list_for_each_safe(lchunk, ltemp, &transport->transmitted) {
+     chunk = list_entry(lchunk, struct sctp_chunk,
+                        transmitted_list);
+
+     /* If the chunk is abandoned, move it to abandoned list. */
+     if (sctp_chunk_abandoned(chunk)) {
+       list_del_init(lchunk);
+       sctp_insert_list(&q->abandoned, lchunk);
+       continue;
+     }
+
+     /* If we are doing retransmission due to a fast retransmit,
+      * only the chunk's that are marked for fast retransmit
+      * should be added to the retransmit queue.  If we are doing
+      * retransmission due to a timeout or pmtu discovery, only the
+      * chunks that are not yet acked should be added to the
+      * retransmit queue.
+        */
+
+     //Added by Simula to allow bundling in fast retransmits:
+
+       if(check_stream_before_add(transport, chunk, fast_retransmit)){
+
+         /*
+         //Old:
+         if ((fast_retransmit && (chunk->fast_retransmit > 0)) ||
+         (!fast_retransmit && !chunk->tsn_gap_acked)) {
+         */
+
+         /* RFC 2960 6.2.1 Processing a Received SACK
+          *
+        * C) Any time a DATA chunk is marked for
+        * retransmission (via either T3-rtx timer expiration
+        * (Section 6.3.3) or via fast retransmit
+        * (Section 7.2.4)), add the data size of those
+        * chunks to the rwnd.
+        */
+       q->asoc->peer.rwnd += sctp_data_size(chunk);
+       q->outstanding_bytes -= sctp_data_size(chunk);
+       transport->flight_size -= sctp_data_size(chunk);
+
+       /* sctpimpguide-05 Section 2.8.2
+        * M5) If a T3-rtx timer expires, the
+        * 'TSN.Missing.Report' of all affected TSNs is set
+        * to 0.
+        */
+       chunk->tsn_missing_report = 0;
+
+       /* Added by Simula:
+
+       Mark first byte in payload of packet according to what
+       caused the retransmission:
+
+       'q' = Not retransmitted yet
+       'f' = Fast retransmit
+       't' = Timeout
+       'b' = Bundling caused by timeout
+       */
+
+       if (sctp_thin_debug_tag_payload){
+         data_chunk_payload = (char *) chunk->skb->data;
+
+         /*
+           for(chunk_offset = 16;
+           chunk_offset < WORD_ROUND(ntohs(chunk->chunk_hdr->length))
+          && data_chunk_payload[chunk_offset] != 'q';
+           chunk_offset++);
+         */
+
+         chunk_offset = 16; // first byte of data chunk payload
+
+         if(fast_retransmit){
+           data_chunk_payload[chunk_offset] = 'f';
+         }
+         else{
+
+           currentTSN = ntohl(chunk->subh.data_hdr->tsn);
+
+           if(currentTSN == lowestTSN){
+             data_chunk_payload[chunk_offset] = 't';
+           }
+           else{
+             data_chunk_payload[chunk_offset] = 'b';
+           }
+         }
+       }
+
+       /* If a chunk that is being used for RTT measurement
+        * has to be retransmitted, we cannot use this chunk
+        * anymore for RTT measurements. Reset rto_pending so
+              * that a new RTT measurement is started when a new
+              * data chunk is sent.
+              */
+             if (chunk->rtt_in_progress) {
+               chunk->rtt_in_progress = 0;
+               transport->rto_pending = 0;
+             }
+
+             /* Move the chunk to the retransmit queue. The chunks
+              * on the retransmit queue are always kept in order.
+              */
+
+             list_del_init(lchunk);
+             sctp_insert_list(&q->retransmit, lchunk);
+           }
+       }
+
        SCTP_DEBUG_PRINTK("%s:␣transport:␣%p,␣fast_retransmit:␣%d,␣"
                          "cwnd:␣%d,␣ssthresh:␣%d,␣flight_size:␣%d,␣"
                          "pba:␣%d\n", __FUNCTION__,
@@ -453,7 +595,7 @@
                          transport->cwnd, transport->ssthresh,
                          transport->flight_size,
                          transport->partial_bytes_acked);
-
+
 }

 /* Mark all the eligible packets on a transport for retransmission and force
@@ -468,9 +610,14 @@
        switch(reason) {
        case SCTP_RTXR_T3_RTX:
                sctp_transport_lower_cwnd(transport, SCTP_LOWER_CWND_T3_RTX);
+
+               /* Added by Simula */
+               update_lost_packets_by_timeout(transport);
+
                /* Update the retran path if the T3-rtx timer has expired for
                 * the current retran path.
                 */
+
                if (transport == transport->asoc->peer.retran_path)
                        sctp_assoc_update_retran_path(transport->asoc);
                break;
@@ -483,8 +630,9 @@
                break;
        }

-       sctp_retransmit_mark(q, transport, fast_retransmit);
-
+         sctp_retransmit_mark(q, transport, fast_retransmit);
+
+
        /* PR-SCTP A5) Any time the T3-rtx timer expires, on any destination,
         * the sender SHOULD try to advance the "Advanced.Peer.Ack.Point" by
         * following the procedures outlined in C1 - C5.
@@ -616,4 +764,3 @@

                /* If we are here due to a retransmit timeout or a fast
```

```
                  * retransmit and if there are any chunks left in the retransmit
-                 * queue that could not fit in the PMTU sized packet, they need
                  * to be marked as ineligible for a subsequent fast retransmit.
+                 * queue that could not fit in the PMTU sized packet, they need
+                 * to be marked as ineligible for a subsequent fast retransmit.
                  */
                 if (rtx_timeout && !lchunk) {
                         list_for_each(lchunk1, lqueue) {
@@ -642,6 +791,118 @@
         return error;
 }
 
+
+/* This routine is added by Simula to bundle outstanding chunks in a new packet if the
+   stream is thin and the MTU allows it. First, it checks if the packet contains
+   chunks that will get retransmitted. If this is true, bundling is not allowed as
+   bundling already has been performed.
+   Then it traverses through the transmitted queue and bundles outstanding
+   chunks by adding them to an outstanding chunk list.
+   At last, the outstanding chunk list is joined with the packet chunk list
+   and the packet size is updated.
+   The first paramater is a pointer to the packet that will be sent.
+   The second paramater is a pointer to the respective transport.
+*/
+
+void bundle_outstanding_chunks(struct sctp_packet *packet, struct sctp_transport *transport){
+
+  size_t packet_size, pmtu, outstanding_chunks_size = 0;
+  struct sctp_chunk *chunk;
+  struct list_head *chunk_list, *list_head;
+  struct list_head outstanding_list;
+  int bundling_performed = 0;
+  __u16 chunk_len;
+
+  if(transport == NULL){
+    return;
+  }
+
+  /* Traverse the packet's chunk list and look for data chunks
+   */
+  list_for_each(chunk_list, &packet->chunk_list){
+    chunk = list_entry(chunk_list, struct sctp_chunk,
+                       list);
+
+    /* Find the first data chunk in packet */
+    if(sctp_chunk_is_data(chunk)){
+
+      if(chunk->has_tsn){
+       /* If the first chunk is a retransmission, do not bundle.
+          Outstanding chunks are already bundled.
+       */
+       return;
+      }
+      else{
+
+       /* If the first data chunk has not been given a TSN yet, then this packet contains
+          new chunks and bundling of outstanding chunks is allowed.
+       */
+       break;
+      }
+    }
+  }
+
+  packet_size = packet->size;
+
+  /* Find the network MTU */
+  pmtu  = ((packet->transport->asoc) ?
+          (packet->transport->asoc->pathmtu) :
+          (packet->transport->pathmtu));
+
+  /* Initiate a linked list that will contain outstanding chunks */
+  INIT_LIST_HEAD(&outstanding_list);
+
+  /* Traverse the transmitted queue and bundle outstanding chunks
+     as long as size of chunks in packet < MTU.
+     Oldest outstanding chunks (First in transmitted queue) is bundled first.
+  */
+
+  list_for_each(list_head, &transport->transmitted){
+
+    chunk = list_entry(list_head, struct sctp_chunk,
+                       transmitted_list);
+
+    if(sctp_chunk_is_data(chunk) && !chunk->tsn_gap_acked){
+      if(chunk->has_tsn){
+
+        /* Chunk length must be a multiple of 4 bytes, pad length if necessary */
+        chunk_len = WORD_ROUND(ntohs(chunk->chunk_hdr->length));
+
+        /* If allowed by MTU, bundle the outstanding chunk by adding
+           it to the outstanding chunks list */
+        if((packet_size + chunk_len) > pmtu){
+          break;
+        }
+        else{
+
+          packet_size += chunk_len;
+          outstanding_chunks_size += chunk_len;
+          chunk->transport = packet->transport;
+          list_add_tail(&chunk->list, &outstanding_list);
+          bundling_performed = 1;
+        }
+      }
+    }
+  }
+
+  /* If bundling has been performed, join the outstanding chunks list at the
+     head of the packet's chunk list and update packet size. This way,
+     all chunks in a packet will be in increasing order, which is requiered
+     by SCTP. Before the packet is transmitted, the data chunks that were
+     originally stored in the packet chunk list will later get TSNs that
+     are larger than the TSNs of the outstanding chunks.
+  */
+
+  if(bundling_performed){
+    list_splice(&outstanding_list, &packet->chunk_list);
+    packet->size += outstanding_chunks_size;
+  }
```

```
+}
+
+
+
+
+
 /*
  * Try to flush an outqueue.
  *
@@ -952,7 +1213,16 @@
                                                 send_ready);
                packet = &t->packet;
                if (!sctp_packet_empty(packet))
-                       error = sctp_packet_transmit(packet);
+
+                 // Simula: Try to bundle outstanding chunks
+
+                  if(sctp_thin_bundling_outstanding){
+                    if(asoc->packets_in_flight < asoc->thin_stream_threshold){
+                      bundle_outstanding_chunks(packet, t);
+                    }
+                  }
+
+                 error = sctp_packet_transmit(packet);
        }

        return error;
@@ -1031,6 +1301,18 @@

        sack_ctsn = ntohl(sack->cum_tsn_ack);

+       /* added by Simula to determine how to count packets in flight
+          based on SACK information and determine packet loss rate: */
+       if(sack->num_gap_ack_blocks > 0){
+
+         update_lost_packets_by_sackgap(asoc, sack);
+         //updated algorithm
+       }
+       else{
+         remove_pkts_in_flight(asoc, sack_ctsn);
+       }
+
+

        /*
         * SFR-CACC algorithm:
         * On receipt of a SACK the sender SHOULD execute the
@@ -1210,6 +1492,10 @@
        __u8 restart_timer = 0;
        int bytes_acked = 0;

+       struct list_head *list_chunk; //Simula
+       struct sctp_chunk *cur_chunk, *oldest_outstanding_chunk; //Simula
+
+

        /* These state variables are for coherent debug output. --xguo */

 #if SCTP_DEBUG
@@ -1510,10 +1796,52 @@
                                        sctp_transport_put(transport);
```

```
                        }
                } else if (restart_timer) {
-                       if (!mod_timer(&transport->T3_rtx_timer,
-                                      jiffies + transport->rto))
-                               sctp_transport_hold(transport);
+
+                 if(sctp_thin_restart_timer){
+
+                   /* added by Simula to restart the timer such that no more than RTO ms
+                      elapse before the timer expires.
+                 */
+                   oldest_outstanding_chunk = NULL;
+
+                   list_for_each(list_chunk, &tlist){
+                     cur_chunk = list_entry(list_chunk, struct sctp_chunk,
+                                            transmitted_list);
+
+                     /* Try to find oldest outstanding chunk */
+                     if(sctp_chunk_is_data(cur_chunk)){
+                       if(!cur_chunk->tsn_gap_acked){
+                         oldest_outstanding_chunk = cur_chunk;
+                         break;
+                       }
+                     }
+                   }
+                   if(oldest_outstanding_chunk != NULL){
+                     /* Subtract age of oldest outstanding chunk
+                        before updating new RTO value.
+                     */
+                     if (!mod_timer(&transport->T3_rtx_timer,
+                                    (jiffies -
+                                     (jiffies - oldest_outstanding_chunk->sent_at)
+                                     + transport->rto))){
+                       sctp_transport_hold(transport);
+                     }
+                   }
+                   else{
+                     if (!mod_timer(&transport->T3_rtx_timer,
+                                    jiffies + transport->rto)){
+                       sctp_transport_hold(transport);
+                     }
+                   }
+                 }// if(sctp_thin_restart_timer)
+                 else{
+                   if (!mod_timer(&transport->T3_rtx_timer,
+                                  jiffies + transport->rto)){
+                     sctp_transport_hold(transport);
+                   }
+                 }
                }
+
        }

        list_splice(&tlist, transmitted_queue);
@@ -1529,6 +1857,12 @@
        struct sctp_chunk *chunk;
        struct list_head *pos;
        __u32 tsn;
+
+        /* added by Simula to hold the missing_report threshold needed to trigger
+           a fast retransmit of a chunk.
+        */
+        int fr_threshold;
+
         char do_fast_retransmit = 0;
         struct sctp_transport *primary = q->asoc->peer.primary_path;

@@ -1568,10 +1902,34 @@
                  * retransmission and start the fast retransmit procedure.
                  */

-                if (chunk->tsn_missing_report >= 3) {
+                /*
+                  if (chunk->tsn_missing_report >= 3) {
+                        chunk->fast_retransmit = 1;
+                        do_fast_retransmit = 1;
+                }
+                */
+
+                /* Added by Simula to trigger fast retransmits after 1 SACK
+                   if stream is thin
+                */
+
+                fr_threshold = 3;
+
+                if(sctp_thin_fr){
+                  if(q->asoc->packets_in_flight < q->asoc->thin_stream_threshold){
+                    fr_threshold = 1;
+                  }
+                  else{
+                    fr_threshold = 3;
+                  }
+                }
+
+
+                if (chunk->tsn_missing_report >= fr_threshold) {
                         chunk->fast_retransmit = 1;
                         do_fast_retransmit = 1;
                 }
+
         }

         if (transport) {
@@ -1741,3 +2099,198 @@
                 SCTP_INC_STATS(SCTP_MIB_OUTCTRLCHUNKS);
         }
 }
+
+/* added by Simula:
+
+   This routine removes all packets from the list which have a TSN lesser than
+   or equal to the cumulative TSN of a SACK. For each removal, packets_in_flight is
+   decremented by one. Packets that have already been counted for must
+   not be counted again. Therefore packets_in_flight is incremented
+   by packets_left_network before packets_left_network is reset to zero.
+   The first parameter is a pointer to the current sctp_association holding
+   the 'packets in flight' datastructure. The second parameter is
+   the cumulative TSN of the SACK.
```

```
+*/
+
+void remove_pkts_in_flight(struct sctp_association *a, __u32 sack_cum_tsn){
+
+
+
+  struct pkt_stat *pkt, *pkt_temp, *pkt_next;
+
+  for(pkt = a->pkt_stat_head; pkt != NULL; pkt = pkt_next){
+
+    if(pkt->highestTSN <= sack_cum_tsn){
+
+      if(!pkt->is_gap_acked) // updated algorithm
+       a->packets_in_flight--;
+
+      if(pkt == a->pkt_stat_head){
+
+       /* Removing head of list */
+       a->pkt_stat_head = pkt->next;
+
+       if(a->pkt_stat_head != NULL){
+          a->pkt_stat_head->prev = NULL;
+        }
+       pkt->next = NULL;
+        kfree(pkt);
+       pkt_next = a->pkt_stat_head;
+      }
+      else if(pkt == a->pkt_stat_tail){
+       /* Removing tail of list */
+       if(pkt->prev != NULL){
+         a->pkt_stat_tail = pkt->prev;
+          pkt->prev->next = NULL;
+       }
+       pkt->prev = NULL;
+        kfree(pkt);
+        pkt_next = NULL;
+      }
+      else{
+       /* Removing an inbetween element */
+       pkt->prev->next = pkt->next;
+        pkt->next->prev = pkt->prev;
+        pkt_temp = pkt->next;
+       pkt->next = NULL;
+        pkt->prev = NULL;
+        kfree(pkt);
+        pkt_next = pkt_temp;
+      }
+
+    }
+    else{
+      pkt_next = pkt->next;
+    }
+  }
+
+}
+
+
+/* Added by Simula:
+    This routine update a loss indication of packets in the packet list
+   after receiving a SACK. For each gap ack block and gap in a SACK
+   the packet list is traversed to update loss indications and
+   mark packets as lost or received.
+*/
+
+void update_lost_packets_by_sackgap(struct sctp_association *a, struct sctp_sackhdr *sack){
+
+  sctp_sack_variable_t *frags = sack->variable;
+  __u32 tsn, sack_ctsn, startBlock, endBlock;
+  int i;
+
+  sack_ctsn = ntohl(sack->cum_tsn_ack);
+
+  tsn = sack_ctsn + 1;
+
+  for(i = 0; i < ntohs(sack->num_gap_ack_blocks); i++){
+
+    startBlock = sack_ctsn + ntohs(frags[i].gab.start);
+    endBlock = sack_ctsn + ntohs(frags[i].gab.end);
+
+    for( ; tsn < startBlock; tsn++ ){
+      loss_indication_update(a, tsn);
+    }
+
+    for( ; tsn <= endBlock; tsn++){
+      mark_gap_acked(a, tsn);
+    }
+
+  }
+}
+
+/* Added by Simula:
+   This routine gap acks a packet in the packet list when a gap acked
+   TSN in a SACK matches the highest TSN in a packet.
+*/
+
+void mark_gap_acked(struct sctp_association *a, __u32 gap_acked_tsn){
+
+  struct pkt_stat *cur;
+
+  for(cur = a->pkt_stat_head; cur != NULL; cur = cur->next){
+
+    if(cur->highestTSN == gap_acked_tsn && !cur->is_gap_acked){
+      cur->is_gap_acked = 1;
+      a->packets_in_flight--; // Updated algorithm
+
+      printk(KERN_ALERT "TSN:␣%u␣is␣gap␣acked,␣mem␣location:␣%x,␣timestamp:␣%d,␣now:␣%d\n",
+           cur->highestTSN, cur, jiffies_to_msecs(cur->timestamp), jiffies_to_msecs(jiffies));
+    }
+  }
+}
+
+/* Added by Simula:
+   This routine marks a packet as indicated lost when a gap in a SACK matches
+   the highest TSN of a packet in the packet list.
+   If the packet is indicated lost by 3 SACKs, the packet is marked lost
+   and lost_packets counter is increased by one.
+
+   Able to fit the requirements to the stream's current thickness
```

```
+*/
+
+void loss_indication_update(struct sctp_association *a, __u32 gap_tsn){
+
+  struct pkt_stat *cur;
+
+  for(cur = a->pkt_stat_head; cur != NULL; cur = cur->next){
+
+    if(cur->highestTSN == gap_tsn && !cur->marked_lost && !cur->is_gap_acked){
+
+      cur->indicated_lost++;
+
+      if(cur->indicated_lost == 3 ||
+        (a->packets_in_flight < a->thin_stream_threshold && cur->indicated_lost == 1)){
+       cur->marked_lost = 1;
+       a->lost_packets++;
+      }
+    }
+  }
+}
+
+/* Added by Simula:
+
+   This routine traverses the packet list after a timeout
+   and marks packets as lost if no SACK has arrived
+   and gap acked the packet since the timeout occured.
+
+   If the packet were sent just before a timeout and had
+   no chance to get a SACK according to the min RTT + 10 ms,
+   the packet is not marked as lost.
+
+*/
+
+void update_lost_packets_by_timeout(struct sctp_transport *t){
+
+  struct sctp_association *a = t->asoc;
+  struct pkt_stat *cur;
+
+  for(cur = a->pkt_stat_head; cur != NULL; cur = cur->next){
+
+    if(!cur->is_gap_acked && !cur->marked_lost &&
+       (jiffies_to_msecs(jiffies - cur->timestamp)
+       > (jiffies_to_msecs(t->min_rtt) + 10))) //&& !is_spurious(t, cur))
+      {
+       cur->marked_lost = 1;
+       a->lost_packets++;
+      }
+  }
+
+}
+/*
+int is_spurious(struct sctp_transport *t, struct pkt_stat *c){
+
+  struct pkt_stat *cur;
+
+  for(cur = c->prev ; cur != NULL; cur = cur->prev){
+
+    if(cur->marked_lost && (c->highestTSN == cur->highestTSN) &&
+       (jiffies_to_msecs(c->timestamp - cur->timestamp) < jiffies_to_msecs(t->min_rtt))){
```

```
+			return 1;
+		}
+
+	}
+	return 0;
+}
+*/
diff -Naur linux-2.6.16.13/net/sctp/protocol.c linux-2.6.16.13-modSCTP/net/sctp/protocol.c
--- linux-2.6.16.13/net/sctp/protocol.c	2006-05-02 23:38:44.000000000 +0200
+++ linux-2.6.16.13-modSCTP/net/sctp/protocol.c	2007-07-16 15:57:59.000000000 +0200
@@ -1069,6 +1069,14 @@
 	/* Initialize handle used for association ids. */
 	idr_init(&sctp_assocs_id);
 
+	/* Added by Simula: Thin stream mechanisms are turned off by default */
+	sctp_thin_fr = 0;
+	sctp_thin_minrto = 0;
+	sctp_thin_bundling_outstanding = 0;
+	sctp_thin_bundling_fr = 0;
+	sctp_thin_expbackoff = 0;
+	sctp_thin_restart_timer = 0;
+
 	/* Size and allocate the association hash table.
 	 * The methodology is similar to that of the tcp hash tables.
 	 */
diff -Naur linux-2.6.16.13/net/sctp/sm_sideeffect.c linux-2.6.16.13-modSCTP/net/sctp/sm_sideeffect.c
--- linux-2.6.16.13/net/sctp/sm_sideeffect.c	2006-05-02 23:38:44.000000000 +0200
+++ linux-2.6.16.13-modSCTP/net/sctp/sm_sideeffect.c	2007-07-16 15:57:59.000000000 +0200
@@ -449,7 +449,23 @@
 	 * maximum value discussed in rule C7 above (RTO.max) may be
 	 * used to provide an upper bound to this doubling operation.
 	 */
-	transport->rto = min((transport->rto * 2), transport->asoc->rto_max);
+
+	//old:
+	//transport->rto = min((transport->rto * 2), transport->asoc->rto_max);
+
+	/* added by Simula:
+	   If thin stream, then avoid exponential backoff
+	   of the retransmission timer.
+	*/
+
+	if(sctp_thin_expbackoff){
+	  if(asoc->packets_in_flight >= asoc->thin_stream_threshold){
+	    transport->rto = min((transport->rto * 2), transport->asoc->rto_max);
+	  }
+	}
+	else{
+	  transport->rto = min((transport->rto * 2), transport->asoc->rto_max);
+	}
 }
 
 /* Worker routine to handle INIT command failure. */
diff -Naur linux-2.6.16.13/net/sctp/sysctl.c linux-2.6.16.13-modSCTP/net/sctp/sysctl.c
--- linux-2.6.16.13/net/sctp/sysctl.c	2006-05-02 23:38:44.000000000 +0200
+++ linux-2.6.16.13-modSCTP/net/sctp/sysctl.c	2007-07-16 16:51:10.000000000 +0200
@@ -1,6 +1,6 @@
 /* SCTP kernel reference Implementation
  * (C) Copyright IBM Corp. 2002, 2004
```

```
- * Copyright (c) 2002 Intel Corp.
+ * Copyright (c) 2002, 2007 Intel Corp.
  *
  * This file is part of the SCTP kernel reference Implementation
  *
@@ -51,6 +51,23 @@
 static long sack_timer_min = 1;
 static long sack_timer_max = 500;

+
+/* Added by Simula: ID for procvariables:
+   The IDs of the other proc variables is found in sysctl.h
+*/
+
+enum{
+
+  NET_SCTP_THIN_FR = 18,
+     NET_SCTP_THIN_MINRTO = 19,
+     NET_SCTP_THIN_BUNDLING_OUTSTANDING = 20,
+     NET_SCTP_THIN_BUNDLING_FR = 21,
+     NET_SCTP_THIN_EXPBACKOFF = 22,
+     NET_SCTP_THIN_RESTART_TIMER = 23,
+     NET_SCTP_THIN_DEBUG_TAG_PAYLOAD = 24
+     };
+
+
 static ctl_table sctp_table[] = {
        {
                .ctl_name       = NET_SCTP_RTO_INITIAL,
@@ -206,6 +223,70 @@
                .extra1         = &sack_timer_min,
                .extra2         = &sack_timer_max,
        },
+
+       /* Added by Simula: Proc variables to turn thin
+          stream mechanisms on and off */
+
+       {
+               .ctl_name       = NET_SCTP_THIN_FR,
+               .procname       = "thin_fr",
+               .data           = &sctp_thin_fr,
+               .maxlen         = sizeof(int),
+               .mode           = 0644,
+               .proc_handler   = &proc_dointvec
+       },
+       {
+               .ctl_name       = NET_SCTP_THIN_MINRTO,
+               .procname       = "thin_minrto",
+               .data           = &sctp_thin_minrto,
+               .maxlen         = sizeof(int),
+               .mode           = 0644,
+               .proc_handler   = &proc_dointvec
+       },
+       {
+               .ctl_name       = NET_SCTP_THIN_BUNDLING_OUTSTANDING,
+               .procname       = "thin_bundling_outstanding",
+               .data           = &sctp_thin_bundling_outstanding,
+               .maxlen         = sizeof(int),
+               .mode           = 0644,
+               .proc_handler   = &proc_dointvec
+       },
+       {
+               .ctl_name       = NET_SCTP_THIN_BUNDLING_FR,
+               .procname       = "thin_bundling_fr",
+               .data           = &sctp_thin_bundling_fr,
+               .maxlen         = sizeof(int),
+               .mode           = 0644,
+               .proc_handler   = &proc_dointvec
+       },
+
+       {
+               .ctl_name       = NET_SCTP_THIN_EXPBACKOFF,
+               .procname       = "thin_expbackoff",
+               .data           = &sctp_thin_expbackoff,
+               .maxlen         = sizeof(int),
+               .mode           = 0644,
+               .proc_handler   = &proc_dointvec
+       },
+
+       {
+               .ctl_name       = NET_SCTP_THIN_RESTART_TIMER,
+               .procname       = "thin_restart_timer",
+               .data           = &sctp_thin_restart_timer,
+               .maxlen         = sizeof(int),
+               .mode           = 0644,
+               .proc_handler   = &proc_dointvec
+       },
+
+       {
+               .ctl_name       = NET_SCTP_THIN_DEBUG_TAG_PAYLOAD,
+               .procname       = "thin_debug_tag_payload",
+               .data           = &sctp_thin_debug_tag_payload,
+               .maxlen         = sizeof(int),
+               .mode           = 0644,
+               .proc_handler   = &proc_dointvec
+       },
+
        { .ctl_name = 0 }
 };

diff -Naur linux-2.6.16.13/net/sctp/transport.c linux-2.6.16.13-modSCTP/net/sctp/transport.c
--- linux-2.6.16.13/net/sctp/transport.c        2006-05-02 23:38:44.000000000 +0200
+++ linux-2.6.16.13-modSCTP/net/sctp/transport.c        2007-07-16 15:57:59.000000000 +0200
@@ -91,6 +91,9 @@
                                SPP_SACKDELAY_ENABLE;
        peer->hbinterval  = 0;

+       /* Added by Simula */
+       peer->min_rtt = 0;
+
        /* Initialize the default path max_retrans. */
        peer->pathmaxrxt  = sctp_max_retrans_path;
        peer->error_count = 0;
@@ -333,8 +336,36 @@
        /* 6.3.1 C6) Whenever RTO is computed, if it is less than RTO.Min
         * seconds then it is rounded up to RTO.Min seconds.
         */
+
+       /*
+       //old:
        if (tp->rto < tp->asoc->rto_min)
-               tp->rto = tp->asoc->rto_min;
+         tp->rto = tp->asoc->rto_min;
+       */
+
+
+       // added by Simula to set minimum RTO = 200 ms if the stream is thin.
+
+       if(sctp_thin_minrto){
+
+         if(tp->asoc->packets_in_flight < tp->asoc->thin_stream_threshold){
+           if(tp->rto < msecs_to_jiffies(200)){
+             tp->rto = msecs_to_jiffies(200);
+           }
+         }
+         else{
+           if (tp->rto < tp->asoc->rto_min){
+             tp->rto = tp->asoc->rto_min;
+           }
+         }
+       }
+       else{
+         if (tp->rto < tp->asoc->rto_min){
+             tp->rto = tp->asoc->rto_min;
+         }
+       }
+
+

        /* 6.3.1 C7) A maximum value may be placed on RTO provided it is
         * at least RTO.max seconds.
@@ -343,7 +374,20 @@
                tp->rto = tp->asoc->rto_max;

        tp->rtt = rtt;
+

+       /* Added by Simula */
+
+       if(tp->min_rtt > 0){
+         if(rtt < tp->min_rtt)
+           tp->min_rtt = rtt;
+       }
+       else{
+         tp->min_rtt = rtt;
+       }
+
+       printk(KERN_ALERT "Min␣RTT:␣%d\n", jiffies_to_msecs(tp->min_rtt));
+
        /* Reset rto_pending so that a new RTT measurement is started when a
         * new data chunk is sent.
         */
```